university of groningen

University Medical Center Groningen

University of Groningen

16th SC@RUG 2019 proceedings 2018-2019

Smedinga, Reinder; Biehl, Michael

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Publication date:
2019

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Smedinga, R., & Biehl, M. (Eds.) (2019). *16th SC@RUG 2019 proceedings 2018-2019*. Bibliotheek der R.U.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 11-09-2023

university of groningen

faculty of science and engineering

computing science

SC@RUG 2019 proceedings

16th SC@RUG 2018-2019

Rein Smedinga, Michael Biehl (editors)

rug.nl/research/bernoulli

SC@RUG 2019 proceedings

Rein Smedinga
Michael Biehl
editors

2019
Groningen

ISBN (e-pub): 978-94-034-1664-9
ISBN (book): 978-94-034-1665-6
Publisher: Bibliotheek der R.U.
Title: 16th SC@RUG proceedings 2018-2019
Computing Science, University of Groningen
NUR-code: 980

About SC@RUG 2019

Introduction

SC@RUG (or student colloquium in full) is a course that master students in computing science follow in the first year of their master study at the University of Groningen.

SC@RUG was organized as a conference for the sixteenth time in the academic year 2018-2019. Students wrote a paper, participated in the review process, gave a presentation and chaired a session during the conference.

The organizers Rein Smedinga and Michael Biehl would like to thank all colleagues who cooperated in this SC@RUG by suggesting sets of papers to be used by the students and by being expert reviewers during the review process. They also would like to thank Femke Kramer for giving additional lectures and special thanks to Agnes Engbersen for her very inspiring workshops on presentation techniques and speech skills.

Organizational matters

SC@RUG 2019 was organized as follows:
Students were expected to work in teams of two. The student teams could choose between different sets of papers, that were made available through the digital learning environment of the university, *Nestor*. Each set of papers consisted of about three papers about the same subject (within Computing Science). Some sets of papers contained conflicting opinions. Students were instructed to write a survey paper about the given subject including the different approaches discussed in the papers. They should compare the theory in each of the papers in the set and draw their own conclusions, potentially based on additional research of their own.

After submission of the papers, each student was assigned one paper to review using a standard review form. The staff member who had provided the set of papers was also asked to fill in such a form. Thus, each paper was reviewed three times (twice by peer reviewers and once by the expert reviewer). Each review form was made available to the authors through *Nestor*.

All papers could be rewritten and resubmitted, also taking into account the comments and suggestions from the reviews. After resubmission each reviewer was asked to re-review the same paper and to conclude whether the paper had improved. Re-reviewers could accept or reject a paper. All accepted papers[1] can be found in these proceedings.

In her lectures about communication in science, Femke Kramer explained how researchers communicate their findings during conferences by delivering a compelling storyline supported with cleverly designed graphics. Lectures on how to write a paper and on scientific integrity were given by Michael Biehl and a workshop on reviewing was offered by Femke.

Agnes Engbersen gave workshops on presentation techniques and speech skills that were very well appreciated by the participants. She used the two minute madness presentation (see further on) as a starting point for improvements.

Rein Smedinga was the overall coordinator, took care of the administration and served as the main manager of *Nestor*.

Students were asked to give a short presentation halfway through the period. The aim of this so-called two-minute madness was to advertise the full presentation and at the same time offer the speakers the opportunity to practice speaking in front of an audience.

The actual conference was organized by the students themselves. In fact half of the group was asked to fully organize the day (i.e., prepare the time tables, invite people, look for sponsoring and a keynote speaker, create a website, etc.). The other half acted as a chair and discussion leader during one of the presentations.

Students were graded on the writing process, the review process and on the presentation. Writing and rewriting accounted for 35% (here we used the grades given by the reviewers), the review process itself for 15% and the presentation for 50% (including 10% for being a chair or discussion leader during the conference and another 10% for the 2 minute madness presentation). For the grading of the presentations we used the assessments from the audience and calculated the average of these.

The gradings of the draft and final paper were weighted marks of the review of the corresponding staff member (50%) and the two students reviews (25% each).

On 2 April 2019, the actual conference took place. Each paper was presented by both authors. We had a total of 20 student presentations this day.

In this edition of SC@RUG students were videotaped during their 2 minute madness presentation and during the conference itself using the video recording facilities of the University. The recordings were published on *Nestor* for self reflection.

[1] this year, all papers were accepted

Website

Since 2013, there is a website for the conference, see **www.studentcolloquium.n**l.

Sponsoring

The student organizers invited Nanne Huiges and Jelle van Wezel as keynote speakers from *BelSimpel*. The company sponsored the event by providing lunch and coffee and drinks at the end of the event.
Hence, we are very grateful to

- BelSimpel

for sponsoring this event.

Thanks

We could not have achieved the ambitious goals of this course without the invaluable help of the following expert reviewers:

- Lorenzo Amabili
- Sha Ang
- Michael Biehl
- Frank Blaauw
- Kerstin Bunte
- Mauricio Cano
- Heerko Groefsema
- Dimka Karastoyanova
- Jiri Kosinka
- Michel Medema
- Jorge A. Perez
- Jos Roerdink
- Jie Tan

and all other staff members who provided topics and provided sets of papers.
Also, the organizers would like to thank the *Graduate school of Science* for making it possible to publish these proceedings and sponsoring the awards for best presentations and best paper for this conference.

Rein Smedinga
Michael Biehl

Since the tenth SC@RUG in 2013 we added a new element: the awards for best presentation, best paper and best 2 minute madness.

Best 2 minute madness presentation awards

2019

Kareem Al-Saudi and Frank te Nijenhuis

Deep learning for fracture detection in the cervical spine

2018

Marc Babtist and Sebastian Wehkamp

Face Recognition from Low Resolution Images: A Comparative Study

2017

Stephanie Arevalo Arboleda and Ankita Dewan

Unveiling storytelling and visualization of data

2016

Michel Medema and Thomas Hoeksema

Implementing Human-Centered Design in Resource Management Systems

2015

Diederik Greveling and Michael LeKander

Comparing adaptive gradient descent learning rate methods

2014

Arjen Zijlstra and Marc Holterman

Tracking communities in dynamic social networks

2013

Robert Witte and Christiaan Arnoldus

Heterogeneous CPU-GPU task scheduling

Best presentation awards

2019

Sjors Mallon and Niels Meima

Dynamic Updates in Distributed Data Pipelines

2018

Tinco Boekestijn and Roel Visser

A comparison of vision-based biometric analysis methods

2017

Siebert Looije and Jos van de Wolfshaar

Stochastic Gradient Optimization: Adam and Eve

2016

Sebastiaan van Loon and Jelle van Wezel

A Comparison of Two Methods for Accumulating Distance Metrics Used in Distance Based Classifiers

and

Michel Medema and Thomas Hoeksema

Providing Guidelines for Human-Centred Design in Resource Management Systems

2015

Diederik Greveling and Michael LeKander

Comparing adaptive gradient descent learning rate methods

and

Johannes Kruiger and Maarten Terpstra

Hooking up forces to produce aesthetically pleasing graph layouts

2014

Diederik Lemkes and Laurence de Jong

Pyschopathology network analysis

2013

Jelle Nauta and Sander Feringa

Image inpainting

Best paper awards

2019

Wesley Seubring and Derrick Timmerman

A different approach to the selection of an optimal hyperparameter optimisation method

2018

Erik Bijl and Emilio Oldenziel

A comparison of ensemble methods: AdaBoost and random forests

2017

Michiel Straat and Jorrit Oosterhof

Segmentation of blood vessels in retinal fundus images

2016

Ynte Tijsma and Jeroen Brandsma

A Comparison of Context-Aware Power Management Systems

2015

Jasper de Boer and Mathieu Kalksma

Choosing between optical flow algorithms for UAV position change measurement

2014

Lukas de Boer and Jan Veldthuis

A review of seamless image cloning techniques

2013

Harm de Vries and Herbert Kruitbosch

Verification of SAX assumption: time series values are distributed normally

Contents

1 Role of Data Provenance in Visual Storytelling
Oodo Hilary Kenechukwu and Shubham Koyal **9**

2 Comparing Phylogenetic Trees: an overview of state-of-the-art methods
Hidde Folkertsma and Ankit Mittal **14**

3 Technical Debt decision-making: Choosing the right moment for resolving Technical Debt
Ronald Kruizinga and Ruben Scheedler **19**

4 An overview of Technical Dept and Different Methods Used for its Analysis
Anamitra Majumdar and Abhishek Patil **25**

5 An Analysis of Domain Specific Languages and Language-Oriented Programming
Lars Doorenbos and Abhisar Kaushal **31**

6 An overview of Technical Dept and Different Methods Used for its Analysis
Anamitra Majumdar and Abhishek Patil **37**

7 Selecting a Logic System for Compliance Regulations
Michaël P. van de Weerd and Zhang Yuanqing **41**

8 Distributed Constraint Optimization: A Comparison of Recently Proposed Complete Algorithms
Sofie Lövdal and Elisa Oostwal **47**

9 An overview of data science versioning practices and methods
Thom Carretero Seinhorst and Kayleigh Boekhoudt **53**

10 Selecting the optimal hyperparameter optimization method: a comparison of methods
Wesley Seubring and Derrick Timmerman **59**

11 Reproducibility in Scientific Workflows: An Overview
Konstantina Gkikopouli and Ruben Kip **66**

12 Predictive monitoring for Decision Making in Business Processes
Ana Roman and Hayo Ottens **72**

13 A Comparison of Peer-to-Peer Energy Trading Architectures
Anton Laukemper and Carolien Braams **77**

14 Ensuring correctness of communication-centric software systems
Rick de Jonge and Mathijs de Jager **83**

15 A Comparative Study of Random Forest and Its Probabilistic Variant
Zahra Putri Fitrianti and Codrut-Andrei Diaconu **88**

16 Comparison of data-independent Locality-Sensitive Hashing (LSH) vs. data-dependent Locality-Preserving hashing (LPH) for hashing-based approximate nearest neighbor search
Jarvin Mutatiina and Chriss Santi **94**

17 The application of machine learning techniques towards the detection of fractures in CT-scans of the cervical spine
Kareem Al-Saudi and Frank te Nijenhuis **99**

18 An Overview of Runtime Verification in Various Applications
Neha Rajendra Bari Tamboli and Vigneshwari Sankar **105**

19 An overview of prospect tactics and technologies in the microservice management landscape
Edser Apperloo and Mark Timmerman **111**

20 Dynamic Updates In Distributed Data Pipelines
S.J. Mallon and N. Meima **117**

Role of Data Provenance in Visual Storytelling

Oodo Hilary Kenechukwu (S3878708) Shubham Koyal (S3555852)

Abstract— 'A picture is worth ten thousand words' is a popular quote by Fred R. Barnard. Thus, the use of images, graphs and data representation in demonstrating our narration is key to making a good storyline. The introduction of graphic demonstration in modern storytelling requires data provenance in order to enhance effective quality images used in telling interesting stories.Visual storytelling simply means telling stories with the use of image media like photographs, videos, symbols or data representation. It has a broad field that cuts across many disciplines, but for the purpose of this paper, we are focusing on visual storytelling that has direct link to data visualization, photographing, artifact, digital imaging and marketing. Data provenance plays significant roles in creating insightful storyline because of its informative attributes and the ability to explicitly explain the origin of the information. An instance of how data provenance can be used in visual storytelling is in creating video game application. During the process of video game design and implementation, data are sourced from many areas using different technologies. These technologies deployed can aid in tracking progresses and algorithms used in the design of the video game through the process of data provenance. The benefit of this is that whenever there is an error on the game application, the designer can fall back to provenance in tracking the procedures involved in the development of the application. Hence, we are looking at different possibilities where data plays role in enhancing visual storytelling. At the end of this paper, one can fathom the use of general purpose visual analytic technologies like Open Provenance Model, Spark streaming and Hadoop in data visualization. One would be able to understand the pros and cons of data provenance in visual storytelling.

Index Terms—Storytelling, Visual storytelling, Data Provenance, Data Visualization.

1 Introduction

Storytelling is synonymous with data visualization especially if it intends to draw attention. However, the methods and process of demonstrating your story visually can be diverse in nature. For ages, people have been telling stories about their experiences and myths but the impact of such stories to the audience is key. For instance, the story of the evolution of man from genus Homo to Homo Sapiens is a good narration of the history of man. This story seems abstract unless is supported with image demos that can aid a better understanding of the narration. From this story, one can deduce the linkage between storytelling and data visualization. The significance of telling stories cannot be overemphasized as stories play major roles in our daily lives.

The art of communicating visually in forms that can be read or looked upon, visual storytelling emphasizes the expression of ideas and emotions through performance and aesthetics[1]. One could imagine how a story would look like when every bit of the storyline is represented with texts and figures only. Definitely, the story would not draw attention. Hence, Visual storytelling is non-trivial in the exploration of digital images and data representation. [2] "Humans are visual creatures in such that we depend a lot on what our eyes tell us". Visual Storytelling with the application of Data Provenance gives the story a new dimension by not only explaining the results to the audience visually but by also exploring the ways with which the data was retrieved and how the results were attained from the raw data. This has allowed researchers to shape their results depending on the audience they are presenting it to. For instance, new geological findings can be presented to the general public with an only text-based presentation, however, it can also be visually presented to another geologist with the maps, graphs, and jargons.

The aim of this paper is to explicitly review the roles which data provenance plays in Visualizing Storytelling and make inference on the use of data visualization in complementing Visual Storytelling. Since data provenance is a method of unveiling the origin of data, we are going to juxtapose its role in visual storytelling to further enhance visual narration. Our work[4] ties the information provided by images, user interactions, and exploration findings into a visual story, and thus it creates a clear connection between them. So, this paper is arranged into six sections. Section two presents the concepts of data provenance and visual storytelling while section three details the applications and methodologies in which data provenance functions in visual storytelling. section four discusses the threat of data provenance in visual storytelling while section five summarizes the subject and make further remarks on the future impacts of data provenance in visual storytelling.

- *Oodo Hilary Kenechukwu is with University of Groningen, E-mail: h.oodo@student.rug.nl.*
- *Shubham Koyal is with University of Groningen, E-mail: s.koyal@student.rug.nl.*

Manuscript submitted on 19 March 2019. For information on how to obtain this document, please contact Student Colloquium of FSE, University of Groningen.

2 Concept of Data Provenance and Visual Storytelling

The role of data provenance in visual storytelling can be conceptualized in different ways. We are focusing on how data provenance can help in enhancing the services of visual storytelling through the Open Provenance Model (OPM) and node analysis. Also, we would critically look at the use of programmable technologies such as Apache Spark and Hadoop in the exploration of visual storytelling. Hadoop and spark are more related to data science especially in MapReduce and large and complex data analysis, although this paper will discuss to an extent some other fields where data provenance can be applied in order to visualize data. Briefly, we discuss how spark and Hadoop technology can be used to visualize data and the details would be discussed in section three.

Apache Spark is a new technology designed for the visualization of big data using a custom rendering engine. Apache spark allows interactive analysis and visualization of big data through its visualization tools such as Matplotlib, ggplot, and D3. This can be done by minimizing query latency to the level of the user's understanding. Spark model allows the users to achieve descriptive and exploratory visualization over big and complex data. The is possible with spark's programming model and interfaces which are very compatible with visualization tools.

On the other hand, Hadoop technology can also play remarkable roles in the visualization of large data by transforming data into valuable insights and exploring it with limitless visual analytics. Apache Hadoop just like the spark has a scalable and distributed objective for the analysis and storage of large and complex dataset. The major component of Apache Hadoop is the Hadoop File System (HDFS). Others are distributed processing framework based on Apache MapReduce and a redundant distributed storage system. The connectivity between Zoomdata, Hadoop HDFS, and SQL on Hadoop technology is what brings about big data analytics and visualization in Hadoop. Zoom data allows users to interact with data visualization by taking the query to the data.

2.1 Data Provenance

Because of the increasing complexity of analytical and data tasks, the aim of analytics software is to devise and construct visual abstractions in addition to multifaceted information so to provide usable options[5]. That is how data provenance functions in visual narrative. Provenance has different meanings across many fields. In visual storytelling, it refers to the tracking of procedures with which the data is acquired and the contents of the results of the information in visual storytelling. The final data is not given the priority but each state of the data is of the same importance as the other. Those who work with provenance sometimes forget that provenance is not a goal or need in achieving better image storytelling, but a technical approach employed in satisfying data needs and goals[6]. Data provenance can be further described as tracks of unveiling data from its origin and detailing the contents of such data. The major characteristics of data provenance are the ability to keep track or archive of the system's input and process it for easy manipulation of the dataset for the provision of the origin and tools of such data. The end goal of data provenance[6] is to assist users in understanding their data. Users may not necessarily need provenance but because of its extensive and broad methodologies especially in query languages, it is well presented as a directed acyclic graph (DAG)[2,7].

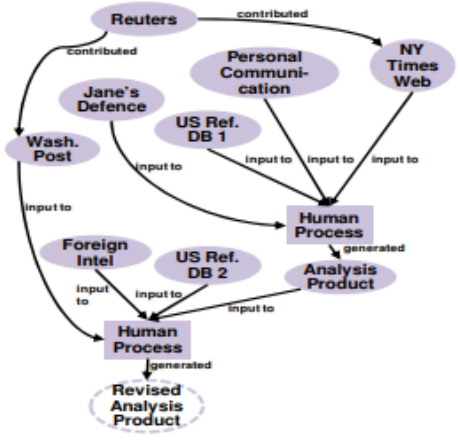


Fig. 1. Example of a Directed Acyclic Graph (DAG)

The provenance graph for a revised analysis product. Public sources (Reuters, Washington Post, New York Times) *Sources* (US Reference Database 1 and 2, those blessed by a particular organization), are also used[7]. There are also personal communications and foreign intelligence sources. Rectangles represent processes while ovals represent data.

A provenance graph is a directed acyclic graph (DAG) shown in Figure 1, $G = (N, E)$, containing a set of nodes, N, and a set of edges, E. Each node has a set of features describing the process or data it represents, e.g., timestamp, description, etc [7]. Edges in the graph denote relationships, such as usedBy, generated, inputTo, etc., between nodes as influenced by OPM [6]. The nodes in the graph above represent data. These data can be referred to any sets of objects. In this case, the nodes may represent raw data, files or arbitrary granularity. The data of provenance may have what is called "breadcrumbs" like identifiers and access which would permit users to have access to the information contents.[14]

2.2 VISUAL STORYTELLING

There is what seems to be a misconception about the term "Visual storytelling" also known as visual narratives. Whereas marketers see it as a marketing tool which aides them in driving their business interests through ads and upturn of word weary spectators, visual arts view it from the point of leeway in making great images ranging from photo shooting to painting. Somewhat, We want to digress into the two analysis for a better understanding of visual storytelling. The analysis of the two views of visual storytelling are as follows:-
In the perspective of visual arts, they conceptualize visual narrative on the aspect of images being filmed or drawn within the frame and how that can influence the attention of the viewers. For us to get a deeper understanding of digital imaging, we can discuss how to make great use of visual storytelling through filming (in photographing for instance). Then, we could be discussing data representation and visualization as tools for visual storytelling. Here, we are introducing imaging for us to determine how lens and light in photographing can affect the images. So, What makes a particular image have a standard over the others? To make a desirable image through filming, drawing or painting, there is a need to consider how to construct the image(s) in a frame. Over the years, artists have been creating visual images with different techniques which are still in use till today. Two techniques that are relevant in this field are *framing* and *composition*. These two practices are relevant in a film just as it is in painting and classic design. Past experience shows how objects can be arranged in the frame using alignment and shapes to make it look attractive, but the use of framing and composition help to guide the eyes towards the direction where the image is located. Meanwhile, there are many regulations guiding filming especially on the area of composition but the most significant among them is the *rule of thirds*[3]. So, with the rule of thirds, the frame is normally divided into three segments, vertically and horizontally. Thus, when an object is placed at the intersecting point of the segment, an attractive image is being created.

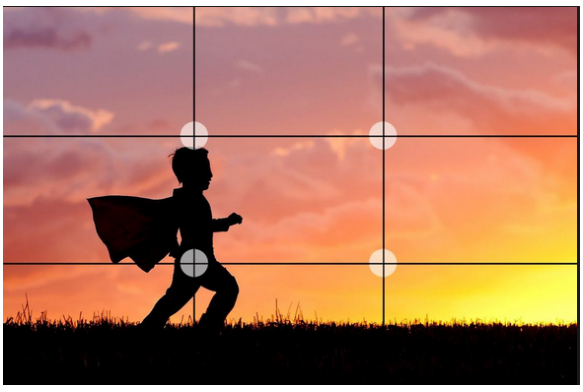


Fig. 2. Rule of Thirds

Figure 2 illustrates the application of the rule of thirds based on three segments of the frame
Source:https://www.photographymad.com/pages/view/rule-of-thirds.

The first person to write a book on the ”rule of thirds” is John Thomas Smith in his book titled ”Remarks on rural scenery”. The part of the book discussed the work of Rembrandt where he said ”Two distinct equal light should never appear in the same picture; One should be principal and the rest subordinate, both in dimension and degree[15]: This can be achieved by keeping the rule of thirds through trope up of the eyes in the frame so the eyes are in focus. By so doing, we draw the attention of the viewers to the image. Still, in composition, we talk about leading lines. Leading line is one of the concepts of composition where lines are used to direct the viewers to where they want them to look. Using this field with the previous rules, one can add to 3-dimensional space. We can also denote the importance of the power of certain objects in a frame, but we can as well use a shallow depth to feel the notes of important matters and character in the story. So, bringing an image closer to the frame makes one know how important the image is, otherwise shrinking the subject matter denotes the feeling or the worst of the scene.

Considering the perspectives of marketers that view visual storytelling as marketing tools, spectators are the key focus here and to draw their attention closer to your products, you need to include in your advertisements those graphics contents of your products so that they can be easily enticed. As Rolf Jensen aptly states, ”We are entering the emotion-oriented *dream society* where customers take for granted the functionality of products and make purchase decisions based upon to what degree they believe a product will give them positive experiences *(storytelling advertising: A visual marketing analysis by Sara Elise Vare)*. On various social media platforms, a lot of products are marketed online through visual narrative. The popularity of some of the big names in information technology was able to be attained by their visual ads. For instance, Amazon which has one the highest product services online sponsors products and services advertisements. Also, Google runs ads called Google shopping Network through which it directs Google searchers on the particular product they are looking for.

2.3 Factors that determine the effectiveness of Visual Story

We are going to look at four components that institute the effectiveness of visual storytelling.

Authenticity and Genuineness It is inherent that when you maintain a standard in your deals, especially as it relates to marketing, prospective users must keenly look for you. You can imagine how consumers react to images of a well-packaged product online; that enthusiasm and zeal to have feelings of the product is a driving tool for making good sales. However, more demands for the products are experienced when there is a steady customary over a period of years. For instance, the security architecture of Apple has made them outstanding that you may not need to create doubt about the end to end encryption of your Apple devices.

Cultural Relevancy and importance Here, we talk about products that are socially recognized. Society can accept a product based on day to day upgrade of the product especially as it reflects the yearnings of the people. For instance, in visual arts, photographs were earlier filmed with analog devices which do not show clear images of the owner, but nowadays, the invention of digital images has made filming more interesting and sharp and clearer images are produced.

Sensory Currency This factor tries to provide immediate solutions to the problems that the audience are desperately looking for.

Portray realistic assumption On this context, your listeners are not supposed to see your products as a function of passion quest.

3 Application of Data Provenance in Visual Storytelling

Data provenance is a special technique that is practically important in tracing the data origin, the complete contents of the data and the actions taken on the data. Relating this to storytelling is a bit tricky. Nevertheless, the role of data provenance in visual storytelling cannot be overemphasized. In many fields, data provenance combined with visual storytelling can bring sizable changes to how data are presented. This is not limited to experts of that field but it can also help a person outside the field understand what the narrator is trying to tell. With the use of visual storytelling to explore complex data, human error can be minimized. Also, data provenance makes revisiting old data quite easy which can shed more light on new findings and misinterpretation of the new data is hugely reduced this way[1].

To understand the application of data provenance in visual storytelling, it can be further discussed in two major areas; data visualization and digital image processing.

3.1 Data Visualization

The role of data provenance in large data visualization was earlier discussed briefly in section two of this paper. Here, this section tends to discuss in details how apache spark and Hadoop metadata technologies can be used to analyze and visualize complex dataset.

Apache Spark The use of spark to visualize exploratory data through the browser is a critical part of data provenance. So, spark plays the role of putting back large dataset into a simple workflow of data analysis via a visual graph. This process is the most effective, scalable, reproducible and distributive method of visualizing data. Also, the spark can visualize expository data through the design of a graph that is modeled into a directed acyclic graph (DAG) of the Open Provenance Model. Then its algorithm can be shared to other users or to the browser for live visualization. There is a saying that a graph is worth more than a thousand words.

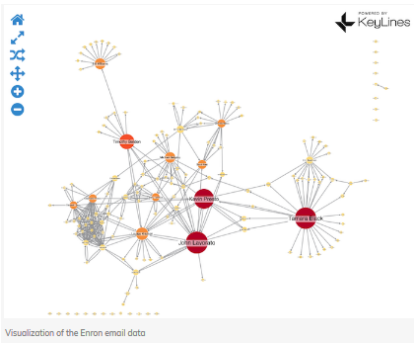


Fig. 3. Graph Visualization of Enron email data. Source: *The Keyline blog*

Graph visualization is a task of presenting visually the active entities that are networked together through the use of nodes. Graphic visualization is said to be one of the most effective and reliable ways of exploring and uncovering the meaning of complex data. So, for the possibility of generating a graph like the one shown above, there are some requirements needed to achieve that. The requirements are;

- Control over details
- Reproducible
- Shareable
- Collaborative
- Interactive

The first two requirements (Control over details and Reproducible) can only be achieved using visualization libraries or programming. The use of computer programming tools like apache spark has many advantages in data visualization because of the grammar expressed in Application Programme Interface (API). Also, data scientist are more flexible in working with such libraries such as matplotlib, D3.js, ggplot, Bokeh, etc. Most of the libraries in use are shareable on the browser and its output can be used on the web as PNG, Canvas, WebGL and SVG. Another significance of using these requirements is that segregating rendering from data manipulation allows the users to work on any of their preferable tools. There are couples of challenges in visualizing complex data. One of the bottlenecks is that it takes a longer time in manipulating a large dataset. Secondly, we have data points than the pixel in the browser. To resolve all these challenges, apache spark streaming is well equipped in solving both problems. For the first problem, spark is very well in managing the CPU and the memory thereby making the system more interactive. The use of spark caching enables the user to take advantage of a memory hierarchy. For the second challenge, it can be resolved by rendering the data. Although there are millions of data points nowadays, spark has techniques to bring down the functionality of data to the level that it can be actually rendered by summarizing, modeling and sampling the data points.

Hadoop Apache Hadoop is one of the applications that are currently in use in managing cluster and MapReduce operations. Hadoop Distributed File System (HDFS) is packaged to run hardware tools. One of the importance of using HDFS is its high scalability features. Just like spark, HDFS has easy access to dataset applications and very suitable for analyzing the large and complex dataset. Some of the achievements of HDFS are;

- Large data Sets: HDFS has capability of running applications with large data sets.It can support millions of files at once with capacity of gigabytes and terabytes files.
- Streaming Data Access: HDFS is primarily packaged for batch processing. So, HDFS application requires streaming access to their data sets.
- Hardware Failure: The large number of server machines acquired by HDFS instance is meant to store system files data.

There are many other data provenance (software) which can help in the visual analysis of data, an example of them is Qlik. Qlik is a business solution software which can enable the users to create an easy and intuitive business reports and analysis.

3.2 Application of Data Provenance in Digital Imaging

In visualizing digital images, we are focusing on the area of documentation process in 2D and 3D digitization through the process of photogrammetry, laser scanning, and similar techniques. Data provenance is required to provide digital image visualization to the end users from diverse backgrounds such as engineering, computer science, and cultural heritage researchers. The end product of 2D and 3D models or orthophotos has multifarious uses ranging from features and geometric analysis to visualization. Image-based data and products should ideally stand as referenceable documents in their own right with a known provenance [9].

Visualization techniques can be applied in medical image analysis in order to access the quantitative and qualitative changes that occurred over the period of time. The role of data provenance in the database is to provide valid and accurate information of the data collected. Medical imaging is essential in providing a channel for the invention of the drugs. Here, we demonstrate how e-science can be deployed in the analysis and experiment of rheumatoid arthritis(RA) model. Globus Toolkit grid software and Virtual Data System can be used to implement visualization of the RA model.

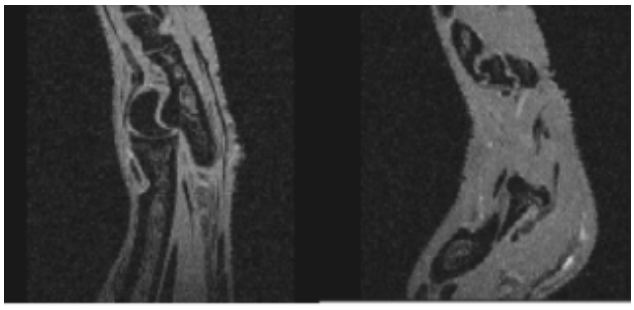


Fig. 4. transaxial view of an ankle Source:*Researchgate; AN IXI EXEMPLER*

Figure 4 shows transaxial view of an ankle at day -12(Left) and at day +13(Right)

The web interface was implemented using Java servlet technology and ran on the Apache Tomcat engine [7]. The essence of this provenance is to have direct query access to the Globus Toolkit database so that the RA image can be visualized. The main web page allowed users to query VDS by the name of transformation and derivation[10].

3.3 Challenges of Data Provenance

Provenance of digital scientific objects is metadata that can be used to determine attribution, to establish casual relationships between objects, to find common tasks parameters that produced similar results as well as for establishing a comprehensive audit trial to assist a researcher wanting to reuse a particular data set[8]. Nevertheless, it requires a lot of efforts and researches to achieve a successful provenance in both visual and data storytelling. In order to achieve this, a lot of provenance questions are required. For instance, according to Shen Xu et al (May 2018), what is the means by which the object in question was created? This question was answered by Macko et al(2013)in this way; introducing local clustering into provenance graphs enables the identification of a significant semantic task through aggregation. Provenance is not used only for digital imaging but also for artworks and marketing. So, data provenance is achieved basically by querying the data (in this case visual storyline) through the use of the provenance graph.

In Figure 5 we can see how provenance can be used to trace the origin , input and processes of an image

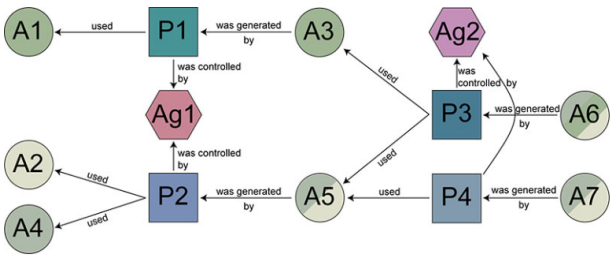


Fig. 5. Trace of data

OPM graph showing the directed structure of the three node types. Artifacts are drawn as circles, Processes are drawn as rectangles and hexagons represent Agents. Source: https://www.provenance.org/

We can view this acyclic graph in reference to the Open Provenance Model (OPM). There are three types of nodes in connection to its basic dependency;

- Artifact: a set of data that has a physical or digital representation of an object in the system.
- Process: the resultant of a new artifact as a result of a series of actions done on the artifact.
- Agent: Conditional operation or individual acting as a catalyst of a process, facilitating, enabling and affecting its execution.

4 Threat of Data Provenance in Data visualization

As useful as Data provenance may sound in visualizing storytelling, it has its own inconveniences. One of the major difficulty with data provenance is the scalability issue. Throughout an operation, there may be multiple updates of the data and keeping tracks of every update may prove to be difficult given the fact that there is no constrains for the number of changes that can be made to the data. Secondly, The need for some levels of control over security incidences is very significant since there are records of attacks to the system vulnerability. So, the introduction of threat modeling in data provenance would help to analyze the security of the systems from the intruders. Moreover, there is a need to have adequate tracking mechanism in place to properly trace the changes in data and represent it correctly. This can prove to be quite costly and also computationally challenging. Although different versions of data are easy to store and reproduce, keeping a track of hows and whats of that version can be problematic in real-world large operations. As the process of storytelling visually can produce large chunks of data, there are large chances of error creeping in the data. As the data gets larger, which is unavoidable in this situation, the chances of error increase exponentially. Keeping the data is a crucial step as it can cause the data further generated to be erroneous. Thus for every data, generating process needs to be checked for errors. This again adds to the cost and also it makes the system slow and sluggish.
The reasons stated above are just the major ones. There exist miscellaneous problems like storage limitations which hugely limits one from using data provenance to its full potential.

5 Conclusion

We have visited the different possible roles of data provenance in visual storytelling. We have discussed cases in which they can be beneficial, for instance in tracing error in visual storytelling and also we have mentioned some situations in which data provenance can be of little use. We have also tried to shed some light on how data provenance can vary from one field to another. Even though the concept of storytelling is as old as mankind but the usage of data provenance to improve visual storytelling is comparatively pretty new and still needs to be perfected upon. We have also hinted the possibilities of using data provenance in visualizing large and complex data. Thus, in the contemporary world, there could be no better story without data provenance considering the increase in the number of data we encounter.

References

[1] Source: Verticalrail (Knowledge base).*What is visual storytelling?* https://www.verticalrail.com/kb/what-is-visual-storytelling/

[2] S.Arevola Arboleda and A. Dewan UNVEILING STORYTELLING AND VISUALIZATION OF DATA Conference: 14th Student Colloquium at University of Groningen

[3] Shen Xu, Tobi Rogers, Elliot Fairweather, Anthony Glenn *From Application of Data Provenace in healthcare analytics software* Proceedings -AMIA

[4] Amabili, L., Kosinka, J., van Meersbergen, M. A. J., van Ooijen, P. M. A., Roerdink, J. B. T. M., Svetachov, P., and Yu, L. (2018). *Improving Provenance Data Interaction for Visual Storytelling in Medical Imaging Data Exploration* In J. Johansson, F. Sadlo, T. Schreck (Eds.), EuroVis 2018 - Short Papers The Eurographic Association.https://doi.org/10.2312/eurovisshort.20181076

[5] Chao Tong 1,*, Richard Roberts 1, Rita Borgo 1 ID , Sean Walton 1 ID , Robert S. Laramee 1, Kodzo Wegba 2, Aidong Lu 2 ID , Yun Wang 3, Huamin Qu 3, Qiong Luo 3 and Xiaojuan Ma 3 *Storytelling and Visualization: An Extended Survey*

[6] Adriane Chapman, Barbara Blaustein,M. David Allen *It's about Data* Provenance as a toll for accessing for accessing data fitness. 4th USENIX workshop.

[7] The Jakarta Site - Apache Tomcat, *http://jakarta.apache.org/tomcat/;*accessed 10-11-2003.

[8] Bechhofer S, Goble C, Buchan I. *Research Objects: Towards Exchange and Reuse of Digital*

[9] N. Carboni, G. Bruseker, A. Guillem, D. Bellido Castaeda et al *Data Provenance in Photogrammetry through documentation process. July, 2016*

[10] Kelvin K.L, Mark Holden, Rolf A.H, Nadeem Saeed, K.J Brooks et al (Use of Data Provenance and the Grid in Medical Analysis and Drug Discovery)

[11] Jimmy Johansson (Contributor), Filip Sadlo (Contributor), Tobias Schreck (Contributor), L. Amabili (Creator), J. Kosinka (Creator), M.A.J. van Meersbergen (Creator), P. M. A. van Ooijen (Creator), J. B. T. M. Roerdink (Creator), P. Svetachov (Creator), L. Yu (Creator)

[12] Eric D. Ragan, Alex Endert, Jibonananda Sanyal, and Jian Chen *Characterizing Provenance in Visualization and Data Analysis: An Organizational Framework of Provenance Types and Purposes*

[13] S. Gratzl1, A. Lex2, N. Gehlenborg3, N. Cosgrove1, and M. Streit1 *From Visual Exploration to Storytelling and Back Again*

[14] Agrawal, R., Imielinski, T., and Swami, A., 1993. Mining association rules between sets of items in large databases. SIGMOD Record, Vol. 22 No. 2

[15] Description of Rule of thirds sourced from https://en.wikipedia.org/wiki/Rule of thirds

Comparing Phylogenetic Trees: an overview of state-of-the-art methods

Hidde Folkertsma, Ankit Mittal

Abstract—Tree-structured data, specifically ordered rooted trees (i.e. trees with a root node, and ordered subnodes for each node), are commonly found in many research areas including computational biology, transportation and medical imaging. For these research areas, comparison of multiple such trees is an important task. The goal of comparing trees is to simultaneously find similarities and differences in these trees and reveal useful insights about the relationship between them. In biology, a prominent example of a comparison task is the comparing of *phylogenetic trees*. These trees contain evolutionary relationships among biological species (their *phylogeny*). In this paper, we compare several methods for the visualization and comparison of phylogenetic trees, and highlight their strengths and limitations. These methods use one or both of two approaches: visual inspection of the data and algorithmic analysis. However, we find that algorithmic analysis loses precision as the trees grow larger. Visual inspection becomes infeasible when the trees grow larger. We show that a combination of both approaches is the most viable.

Index Terms—Phylogenetic trees, ordered rooted trees, data analysis, visual interaction

1 Introduction

Efficient and effective comparison of hierarchical structures such as trees is an important but difficult task in many research areas. In the field of biology, specifically bioinformatics, one such task is the comparison of *phylogenetic trees* [16]. These are ordered rooted trees representing evolutionary relationships among biological species, and are often inferred using the sequenced genomic data of a certain species. Comparing phylogenetic trees can yield useful insights into similarities and differences between species. However, since the trees tend to be very large (more than 500 *taxa*; nodes in the tree representing a taxonomic group), visualizing them becomes an increasingly difficult task. Moreover, comparing two large trees poses an even larger problem. This problem has lead to a need for tools that can both visualize and compare phylogenetic trees effectively.

There are several approaches to this problem [4]. First, a purely algorithmic approach, yielding one or more metrics that indicate similarities and differences between trees. Second, methods that rely on the domain expert's visual inspection, that is, not calculating the similarity but visualizing the trees in a manner that enables the domain expert to spot differences and similarities. Finally there are methods that combine the previous two, in order to use one approach's strengths to combat the other's weaknesses.

Currently, there are many methods available but there is a lack of a clear overview of types of methods' properties. In this paper, we therefore provide an overview of a number of phylogenetic tree comparison methods, and highlight their strengths and limitations.

2 Background

In order to provide a better understanding of the subject, we will briefly describe some concepts related to the phylogenetic tree comparison task. We will also briefly discuss the main challenges researchers face in phylogenetic tree comparison.

2.1 Ordered rooted trees

In graph theory, trees are graphs that are undirected and contain no cycles. In computer science, trees are typically *rooted*, meaning that one of the nodes in the tree is marked as the root node, as shown in figure 1. The root node is the base of the tree, usually shown at the top. It can have several *child nodes* (A and B). Child nodes are called *leaves* of the tree if they don't have any child nodes themselves. In figure 1, A and B are leaves.

An *ordered* tree is a tree where the order of the children of a node is significant [14]. The trees in figure 1 are examples. T_1 and T_2 contain the same data, but the leaves are flipped in T_2. Therefore the trees are not the same if they are ordered trees, because the ordering of the child nodes is different.

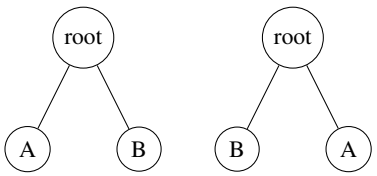


Fig. 1: Left: T_1, right: T_2. If T_1 and T_2 are ordered, $T_1 \neq T_2$, otherwise $T_1 = T_2$.

2.2 Phylogenetic trees

Phylogenetic trees are both ordered and rooted, and represent evolutionary relationships among organisms. The branching in these trees indicates how species evolved from (a series of) common ancestors. An example of a phylogenetic tree is shown in figure 2. In this example, the root node is "Vertebrates", and the leaf nodes are present-day species. Nodes with a more recent (i.e. farther up the tree) common ancestor are more closely related than nodes with a common ancestor less recent common ancestor. From figure 2, we can deduce that lungfish are more closely related to coelacanths than they are to teleosts.

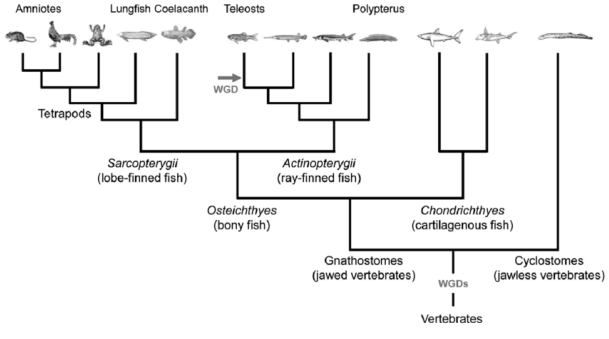


Fig. 2: A phylogenetic tree [18].

- *Hidde Folkertsma is a MSc. Computing Science student at the University of Groningen. E-mail: h.folkertsma.1@student.rug.nl.*
- *Ankit Mittal is a MSc. Computing Science student at the University of Groningen. E-mail: a.mittal@student.rug.nl.*

Nodes in phylogenetic trees generally have exactly 2 children, unless there is an uncertainty about the branching order in that part of the tree. Therefore it is possible to encounter phylogenetic trees that have nodes with 3 or more children.

2.3 Inference of phylogenetic trees

The previous subsection may raise the question how phylogenetic trees are obtained. Historically this has been done by analyzing characteristics of the species and constructing the tree based on characteristics shared by species [1]. Examples of such characteristics are shown in figure 2, e.g. "tetrapod" and "ray-finned".

However, in more recent years, a more popular and accurate tree-inferring approach is the use of DNA sequencing [10]. Using the differences in DNA sequences of homologous genes in different species, the evolutionary relationships between those species can be determined.

2.4 Shortcomings of phylogenetic trees

In using phylogenetic trees, an important underlying assumption is made, namely that evolution always occurs in a tree-like manner. This is, however, not always the case, as *horizontal gene transfer* may occur. Horizontal gene transfer is the transfer of genes from an organism A to an organism B that is not its offspring. B may therefore obtain genetic traits from an ancestor of A that is **not** an ancestor of B. This way of gene transfer is especially common in bacteria. A visualization of this is shown in figure 3.

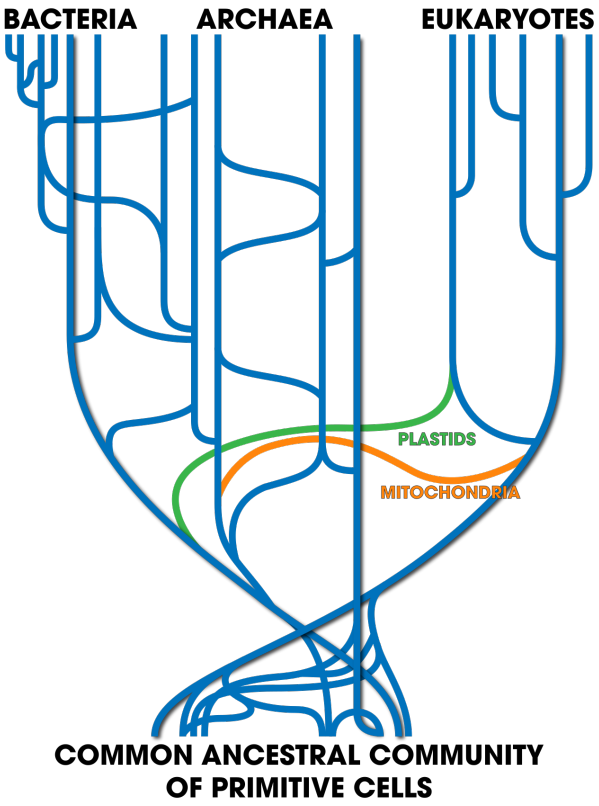


Fig. 3: Visualization of horizontal gene transfer [3]. The horizontal crossovers indicate horizontal gene transfer.

Another shortcoming of phylogenetic trees is that they are only as good as the inference method used to obtain them. This means they don't necessarily reflect the true evolutionary relationships, as the inference method may have produced errors in the tree. This is why comparison is important, because comparing the phylogenetic trees produced by an inference method A to trees produced by some reference inference method B can be useful in evaluating if the inference of A's trees was correct. We can compare inference methods by comparing their produced trees.

2.5 Data comparison

Comparison of data comes with its own challenges. A common problem when performing exploratory analysis is that the researcher wants to find out if there is some relation or difference between the data that is to be compared, but is not sure what to look for. It is therefore hard to derive an algorithm for this task. Another problem is the visual inspection of large trees; this is too time-consuming for a human and therefore requires an algorithm, which in turn suffers from the aforementioned problem regarding the algorithm derivation.

3 Related Work

In order to compare approaches to the phylogenetic tree comparison task, we performed a literature study and found several methods of comparing phylogenetic trees. These methods are either algorithmic approaches, calculating a similarity (or difference) metric, or rely on visual inspection, or combine these two.

3.1 Insights by Visual Comparison: The State and Challenges

Von Landesberger [4] proposes not a phylogenetic tree comparison method specifically, but a more general framework for data comparison tasks. This method advocates an iterative approach, combining algorithmic analysis with visual inspection. A visual depiction of this process is shown in figure 4.

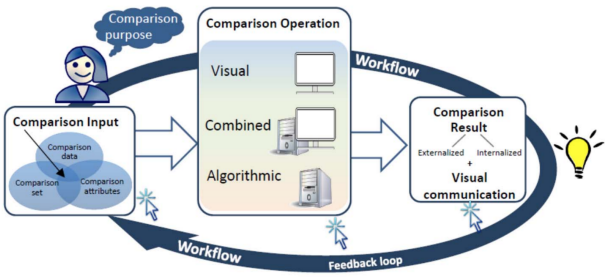


Fig. 4: The visual-analytical comparison process proposed in [4].

The process can be divided into 5 steps:

1. The comparison *purpose*. The user should define a goal of the comparison, e.g. finding significant similarities or differences in trees.

2. The comparison *data*. The data on which the comparison is performed, in this case, a set of phylogenetic trees.

3. The comparison *operator*. This is a very important component, as it determines the outcome of the comparison workflow. It is up to the user to choose the kind of approach to use here.

4. The comparison *result* and its visualization. This visualization step is important for the iterative process. The results of the comparison should be clearly shown to the user of the workflow, in order to allow for better results in a following iteration.

5. The comparison *workflow*. A comparison may not be as simple as comparing tree A to tree B, it may be more complex, e.g. a series of comparisons.

Each step comes with its own challenges, particularly steps 3-5. An algorithmic approach in step 3 requires a suitable and efficient algorithm, which outputs a suitable similarity metric. Visual inspection poses the problem of scalability; screen size and the user's cognitive capabilities are limiting factors. Step 4 poses the same problem, especially when the both the input and output need to be displayed. As

the proposed process is iterative, new insights may be gained as the iterations go on, requiring new comparison workflow steps. The user needs to edit his comparison workflow accordingly.

3.2 Metrics of Phylogenetic Tree similarity

Algorithmic approaches to tree comparison often output a scalar metric indicating the similarity between trees. The similarity indicates the "distance" from tree A to tree B. A distance of 0 would mean that $A = B$. A high distance between two trees indicates that they are significantly different.

3.2.1 Robinson-Foulds distance

An important metric still used in many comparison methods today [8] was proposed back in 1981 by Robinson and Foulds [11]. It is a relatively simple metric that is fast to compute, having implementations in O(n) [9], n being the amount of nodes in the trees. Despite being commonly used, it suffers from a few shortcomings. An important shortcoming is that the algorithm outputs the maximum value fairly quickly, even for trees that are reasonably similar. It is therefore hard to tell if a tree is slightly different or very different based on only this metric. Moreover, it can be imprecise when compared to other methods. Furthermore, moving a leaf in the tree changes the distance score more than moving both the leaf and its immediate neighbour. Robinson-Foulds also assigns a lower distance score to trees that contain more uneven partitions. Balanced trees therefore get lower distances than asymmetric trees [13].

3.2.2 A Metric on Phylogenetic Tree Shapes

Colijn et al. [7] propose an algorithm that considers not a full tree, but only its shape. The shape of a tree is the tree without tip labels and branch lengths. This method considers all possible sub-tree shapes, labels these, and compares these subtrees to compute a final metric. Some results of the method are shown in figure 5, where it is clearly visible that the metric is able to separate two types of flu viruses (one from a tropical origin, the other from the United States).

Despite achieving reasonable results, the method is very costly to compute. A tree with 500 tips generated labels with over 1 million digits, making the method very slow. This was solved by hashing the labels, but with a decently large tree the amount of sub-tree shapes is so large that it exceeds the amount of hashes possible for the used hasing method.

Besides the implementation of their metric, the authors also explain the appeal of metrics in general; they result a single scalar value, which is simple to work with and easy to interpret. However, they are not always efficient to compute for large trees, and for a larger tree, the metric is less capable of describing the similarity. Moreover, with the dropping cost of DNA sequencing, the size of the trees inferred from it will only grow. Therefore scalar metrics in general seem to be getting less effective; they simply cannot capture enough of the information in these large trees.

3.3 Phylo.io

Phylo.io [12] (accessible at `phylo.io`) is a web application specifically developed for the comparing of phylogenetic trees. There are many tools for visualization of phylogenetic trees, but most of them have significant drawbacks. In particular, they lack scalability and therefore cannot deal well with large trees. Attempting to visualize large trees in these tools quickly leads to poor legibility of the resulting visualizations. Moreover, several tools are outdated in terms of its implementation, requiring legacy systems to run. Finally some tools are simply not available.

The drawbacks to existing tools drew the authors of this paper to create Phylo.io, which is web-based and therefore accessible on every machine. More importantly, it was designed specifically for the comparison task. The tree comparison score proposed in [16] is used to indicate the degree of similarity between the trees. Furthermore, it aims to maximize legibility by automatically adjusting the displayed tree to the screen size by collapsing subtrees. Moreover, Phylo.io can automatically compute the internal node in tree B that best corresponds with tree A.

3.4 Using motifs for visual analysis

Landesberger et al. [17] propose a method using motifs in visual analysis of large graphs. Motifs could be defined as patterns of interconnections occurring in complex structures (e.g. trees) at numbers that are significantly higher than those in randomized structures. There are already several tools present that offer motif visualization in analysis of trees, for example - MAVisto, FANMOD and SNAVI. Even though these tools allow fast detection of motifs, are computationally intensive and display found motifs with their frequencies, they have their own limitations [17]. The biggest drawbacks of these tools are they work on simple pre-defined motifs and restrict themselves to visualization of small trees.

These drawbacks encouraged the authors to design their noteworthy approach of user-defined motifs in visual analysis of trees. In this system, the users can define their own motif definition and perform visual analysis on their occurrence and location (see figure 6).

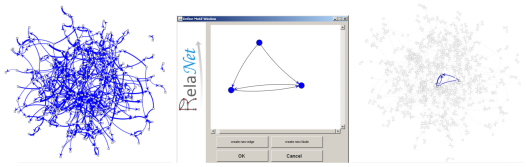


Fig. 6: Interactive motif definition (center) and visualization of found user-defined motif with labeled names of persons (right). Original graph is shown on the left side [17].

Furthermore, the set of motifs that have been found can be filtered in order to focus on structures obeying certain constraints (see figure 7). The authors present their approach for graph aggregations using motifs which reveals higher level structures in the trees (see figure 8).

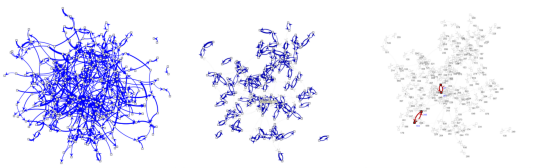


Fig. 7: Motif filtering example [17].

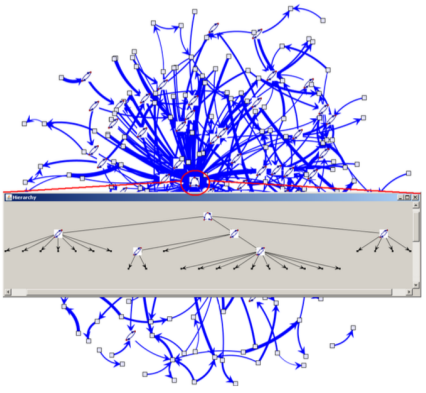


Fig. 8: Aggregation of a node [17].

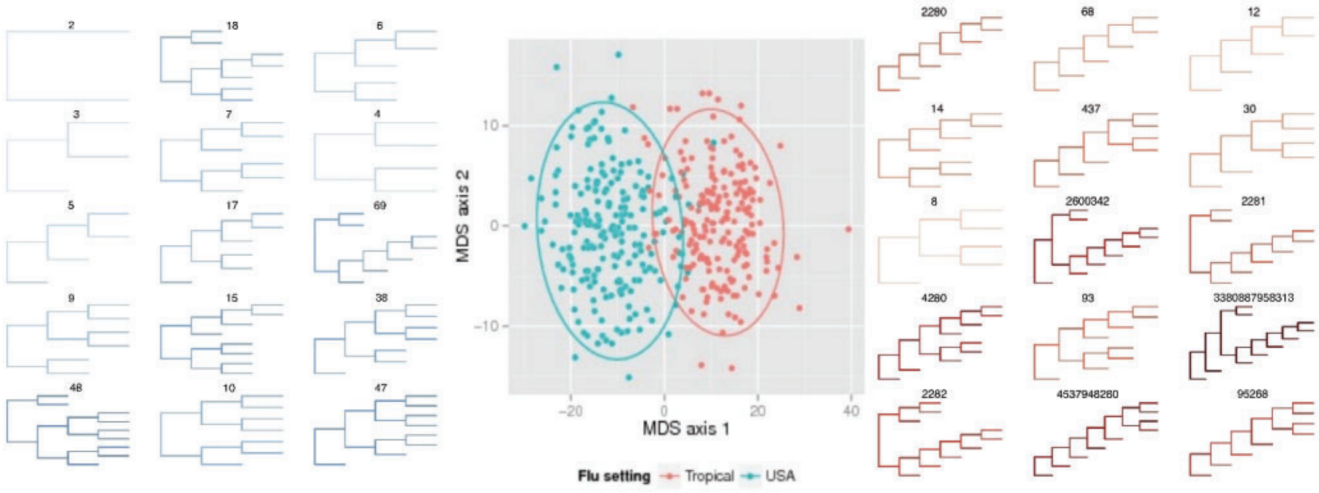


Fig. 5: Comparisons between phylogenetic trees from two types of H3N2 flu virus samples; trees from the two types are separated by the proposed metric [7].

The motif-based approach has been tested for directed trees on a phone call network data set and it has proven efficient. This approach could very well be applied on phylogenetic trees in determining the common pattern of trees (motifs) among different species.

3.5 OInduced

In this paper, OInduced is proposed [5]. OInduced is an algorithm for finding frequent tree patterns in ordered rooted trees and shows the comparison of OInduced algorithm with the already well known algorithms like FREQT [2], iMB3Miner [15]. A tree is called frequent if its per-tree support (occurrence-match support) is more than or equal to a user-specified per-tree (occurrence-match) minsup value. The already present algorithms have certain limitations, for example - FREQT uses an occurrence based approach for finding frequent tree pattern which makes the algorithm inefficient for dense data sets.

OInduced

Require: a database D consisting of rooted ordered labeled trees, a user defined $minsup$ (either *per-tree* or *occurrence-match*).
Ensure: All frequent induced tree patterns.
$Output \leftarrow \emptyset$.
$F1_SET \leftarrow$ the set of all frequent nodes and their encodings.
$F2_SET \leftarrow \emptyset$.
while $F1_SET \neq \emptyset$ **do**
  for all $P_k \in F1_SET$ **do**
    $Ext \leftarrow$ **Extend**(P_k).
    for all $P_{k+1} \in Ext$ **do**
      if $support(P_{k+1}) \geq minsup$ **then**
        $F2_SET \leftarrow F2_SET \cup P_{k+1}$.
      end if
    end for
  end for
  $Output \leftarrow Output \cup F1_SET$.
  $F1_SET \leftarrow F2_SET$.
  $F2_SET \leftarrow \emptyset$.
end while
return $Output$.

Fig. 9: Pseudocode of OInduced [5].

To overcome the drawbacks of existing algorithms, the authors developed their novel algorithm: OInduced (see figure 9), which on high-level is divided into 2 parts. 1. Candidate Generation Method: In this method, the algorithm ensures that the new candidate could only be extended by the only known frequent tree patterns. 2. Frequency Counting: It basically counts the frequency of trees. It is done by two tree encoding: M-coding and Cm-coding which are both combined depth-first/breadth-first traversals (see figure 10).

Encoding

Require: an input tree T.
Ensure: *m-coding* and *cm-coding* of nodes of T.
$mid \leftarrow 0$.
m-coding(root(T))$\leftarrow 0$.
for all nodes x in preorder traversal of T **do**
  for all children r of x in right-to-left order **do**
    $mid \leftarrow mid + 1$.
    m-coding$(r) \leftarrow mid$.
  end for
  cm-coding$(x) \leftarrow mid$.
end for
return *m-coding* and *cm-coding*.

Fig. 10: Pseudocode of m-coding and cm-coding [5].

The authors claim that they have tested their algorithm along with the few existing algorithms like FREQT and iMB3Miner on real data sets as well as synthetic data sets and their algorithm is significant than other algorithm as it reduces the run time and scales linearly with respect to the size of input trees.

4 Conclusion

Comparison of ordered rooted trees is a crucial requirement for the phylogenetic trees. It offers the biologists the chance to verify the validity of their hypothesis, to share the proceedings of their hypothesis and to further develop them. In this paper, we firstly explained about the phylogenetic trees and their shortcomings. Following that, some previously developed methods for comparing the phylogenetic trees are mentioned and described. We mainly focused on the strengths and limitations of each method and tried to give some potential ideas.

We have reviewed several phylogenetic tree comparison methods. Several methods [6, 9, 11] approach the problem in an algorithmic manner, providing the user with a simple output of a scalar metric. However, as trees grow larger, the metric becomes harder to compute and less indicative of the actual similarity of the trees [6].

Other tools use visualization, in these methods the key is how to provide the user with the most interesting (i.e. differing sections) parts of the trees. Oftentimes, this still requires an algorithmic solution.

Concluding, we see that state-of-the-art methods combine algorithmic analysis with visual analysis. Doing this in iteration is a effective way to acquire insights about the tree data [4].

5 Future Work

As science and technology are evolving, new efficient methods will be discovered for the comparison of the phylogenetic trees. Future researcher should possibly focus on designing new methods of comparing phylogenetic trees by probably incorporating the strengths of previously designed methods and eliminating their drawbacks.

6 Acknowledgements

We would like to thank our expert reviewers Lorenzo Amabili and Jiri Kosinka and our colleagues for their valuable feedback.

References

[1] Building the tree. https://evolution.berkeley.edu/evolibrary/article/0_0_0/evo_08. Accessed on February 28th, 2019.

[2] T. Asai, K. Abe, S. Kawasoe, H. Sakamoto, H. Arimura, and S. Arikawa. Efficient substructure discovery from large semi-structured data. *IEICE Transactions*, 87-D(12):2754–2763, Apr 2004.

[3] Barth F. Smets. Tree of life showing vertical and horizontal gene transfers. https://en.wikipedia.org/wiki/Horizontal_gene_transfer. Accessed on February 28th, 2019.

[4] S. Bremm, T. von Landesberger, M. He, T. Schreck, P. Weil, and K. Hamacherk. Interactive visual comparison of multiple trees. In *2011 IEEE Conference on Visual Analytics Science and Technology (VAST)*, pages 31–40, Oct 2011.

[5] M. H. Chehreghani, M. H. Chehreghani, C. Lucas, and M. Rahgozar. Oinduced: An efficient algorithm for mining induced patterns from rooted ordered trees. *IEEE Trans. Systems, Man, and Cybernetics, Part A*, 41(5):1013–1025, Jan 2011.

[6] C. Colijn and M. Kendall. Mapping Phylogenetic Trees to Reveal Distinct Patterns of Evolution. *Molecular Biology and Evolution*, 33(10):2735–2743, Jun 2016.

[7] C. Colijn and G. Plazzotta. A Metric on Phylogenetic Tree Shapes. *Systematic Biology*, 67(1):113–126, May 2017.

[8] K. G. D. Bogdanowicz. Visual treecmp. https://eti.pg.edu.pl/treecmp/. Accessed on February 28th, 2019.

[9] W. H. E. Day. Optimal algorithms for comparing trees with labeled leaves. *Journal of Classification*, 2(1):7–28, Dec 1985.

[10] G. J. Olsen, H. Matsuda, R. Hagstrom, and R. Overbeek. fastDNAml: a tool for construction of phylogenetic trees of DNA sequences using maximum likelihood. *Bioinformatics*, 10(1):41–48, Feb 1994.

[11] D. Robinson and L. Foulds. Comparison of phylogenetic trees. *Mathematical Biosciences*, 53(1):131 – 147, Feb 1981.

[12] O. Robinson, C. Dessimoz, and D. Dylus. Phylo.io : Interactive Viewing and Comparison of Large Phylogenetic Trees on the Web . *Molecular Biology and Evolution*, 33(8):2163–2166, Apr 2016.

[13] M. R. Smith. Bayesian and parsimony approaches reconstruct informative trees from simulated morphological datasets. *Biology Letters*, 15(2), Feb 2019.

[14] R. Stanley. *Enumerative Combinatorics*. Cambridge Studies in Advanced Mathematics. Cambridge University Press, 2012.

[15] H. Tan, F. Hadzic, T. S. Dillon, E. Chang, and L. Feng. Tree model guided candidate generation for mining frequent subtrees from xml documents. *ACM Trans. Knowl. Discov. Data*, 2(2):9:1–9:43, Jul 2008.

[16] T. von Landesberger. Insights by visual comparison: The state and challenges. *IEEE Computer Graphics and Applications*, 38(3):140–148, May 2018.

[17] T. von Landesberger, M. Görner, R. Rehner, and T. Schreck. A System for Interactive Visual Analysis of Large Graphs Using Motifs in Graph Editing and Aggregation . *Vision Modeling and Visualization*, Jan 2009.

[18] K. Yamamoto, S. Bloch, and P. Vernier. New perspective on the regionalization of the anterior forebrain in osteichthyes. *Development, Growth Differentiation*, 59:175–187, May 2017.

Technical Debt decision-making: Choosing the right moment for resolving Technical Debt

Ronald Kruizinga and Ruben Scheedler, University of Groningen

Abstract— Technical debt, software development compromises in maintainability to increase short term productivity, is an often discussed topic with respect to decision-making, due to its prevalence and accompanying costs. Typically taking up 50-70% of a project's time [18], maintenance should be candidate one when it comes to lowering project cost. Many approaches for handling technical debt and deciding when to tackle it are available. These can be used in an agile context but are not restricted to it. In this paper we perform a literature review of four approaches to decision-making with respect to technical debt.
The first approach we discuss consist of the Simple Cost-Benefit analysis, which has its roots in the financial sector. The second approach describes the Analytic Hierarchy Process often found in decision-making literature. The next approach provides an evaluation for the key factors that play a role in whether the debt should be paid immediately or be deferred. The final approach is a highly formalized decision evolution approach. Each of the considered approaches have their own advantages and disadvantages. Over the course of this paper we explain how these methods work and we discuss the strengths and weaknesses of each method. We compare them on seven factors, amongst which the feasibility of adoption in a company and the accuracy and completeness of the approach. Based on this, we find that the Analytic Hierarchy Process is the most feasible approach, but case studies and further research are required for all approaches.

Index Terms—Technical debt, Decision making, Cost analysis, Software systems, Formalized evolution

1 Introduction

Technical debt (TD) was first mentioned by Ward Cunningham in 1992 [2]. It can be defined as *software development compromises in maintainability to increase short term productivity*. Cunningham compares TD to debt and interest as used in the financial sector. Similar to financial debt, TD comes with an increasing interest cost. Companies typically have change management teams that determine what changes are to be included in the next iteration/evolution of a system. An iteration then consists of a set of changes to the system also known as evolution items. These can be new features, bugfixes, workarounds, refactorings and more.

TD can have different causes. It can be caused by complex requirements, lack of skill of the developer, lack of documentation or simply by choice by taking the easy solution instead of the proper one, all of which impacts the way TD should be handled.

The problem of TD becomes clear by considering the size of TD in existing software [3]. Curtis et al. found that on average $3.61 of TD exists for every line of code in over 700 large scale applications [3]. Nugroho et al. estimated during a case study a TD interest of 11 percent in a large scale application, which would grow to 27 percent in 10 years [10]. The impact of interest becomes apparent given that that maintenance activities consume $50-70\%$ of typical project development [18].

Agile working methods have gained more and more traction over the last few years [5]. Although a variety of methods (SCRUM, XP, FDD) is available and refactoring is already an integrated part of these methods [6], none offer a good solution for managing technical debt. Technical Debt Managing (TDM) requires the answer to one question, which is the research question of this paper: How do you determine the right moment to fix TD?

Managing TD principally comes to down to a trade-off between a robust future-proof product that takes longer to develop and a quickly finished product which is hard to maintain.

- *Ronald Kruizinga is a Computing Science master's student at Rijksuniversiteit Groningen. E-mail: r.m.kruizinga@student.rug.nl.*
- *Ruben Scheedler is a Computing Science master's student at Rijksuniversiteit Groningen. E-mail: r.j.scheedler@student.rug.nl.*

In this paper we explore different ways of making this trade-off. We first discuss related work done on the topic of TD decision-making, before elaborating on several approaches.

We start with a simple cost-benefit trade-off, followed by an approach sourced in decision-making literature. We then go over two more complex mathematical approaches, one that quantifies the factors that should be taken into account, such as code metrics and customer satisfaction and another approach that highly formalizes TD decision-making by quantifying TD.

In the discussion we compare these four approaches while placing them in a wider context to find whether there exists a right moment to fix TD and when to fix it. We then discuss our results and whether they are accurate enough. Finally, we provide our conclusion to what the optimal approach is and suggest further research that should be done.

2 Related Work

Martini et al. performed a case study on a subtype of TD: Architectural Technical Debt (ATD) [8]. They interviewed several types of actors in the software development process (architects, developers, scrum masters) and found that ATD comes with objectionable consequences like vicious circles, in which fixing the ATD in a quick manner brings even more ATD.

They were not the first to establish the danger of TD. A growing amount of studies is being performed on the subject of TD. Managing TD mainly consists of three procedures.

- grouping
- quantifying
- decision making

Grouping is the process of translating a system into actionable units. Most related studies focus on TDM in an agile environment [1, 7, 12]. Therefore, we see a general trend in managing TD using a backlog similar to the one used in agile development for feature issues but instead containing technical small independent debt items [12]. This item log takes care of the grouping process.

Quantifying is the process of attaching values to TD items based on certain metrics. Li et al. [7] performed a mapping study towards the current understanding of TD and TDM. It reviews over 90 papers on the topic of TD(M) and gives an overview of approaches used in managing TD. For this they use TD dimensions introduced by Tom et

al. [17]. This research distinguishes different dimensions of TD among which code, architecture and environment. Code debt manifests itself in the form of poorly written code, such as hacks and workarounds. Architectural debt can be found in the form of code design that does not focus enough on maintainability or adaptability. Enviromental debt consists of processes that can be automated or using outdated components/technologies. Li et al. extend the dimensionality into subtypes of TD [7]: requirements, architecture, code, documentation, versioning and more. Their overview of tools used to manage TD shows a major focus solely on code TD. Integrated development environments, for example, offer refactoring for poorly written code.

Interestingly, Nord et al. conducted a research in finding a metric to express TD [9] and although available tools focus primarily on code TD, this research found that code level metrics are insufficient to express TD. It proposes to focus more on architecture-related debt to help optimize releases, which requires to look to at non-functional properties like complexity and performance.

Although tools exist for managing code TD, other types are harder to manage. This is especially clear when it comes to decision-making: given a list of feature items and TD items, which should be selected for the following release?

Seaman et al. [13] propose to measure TD debt in terms of interest and principal. Interest is the extra time required to achieve changes in the system by incurring TD. The principal is the cost (in time) to fix the TD. A second approach that is introduced here is Analytic Hierarchy Process, which is grounded in decision-making literature. We will elaborate on these methods in Sect. 3.1 and Sect. 3.2.

Ho et al. propose a similar process [4], which uses documentation, testing, maintainability and complexity as metrics to quantify TD. They embed the monitoring of these dimensions in a process that weighs features against paying off TD.

Nord et al. propose a somewhat different mathematical model for the task [9]. A model is proposed in which again implementation cost is taken into account, but now combined with the amount of dependencies a (new) component has. The dependencies are the key to determine whether to refactor or whether to postpone it.

Martini et al. introduces an impressive collection of properties to be used to prioritize items [8]. Most interestingly it adds a commercial perspective to the decision-making process by introducing aspects like long-term customer satisfaction and competitive advantage.

Snipes et al. [15] introduce a formal method of selecting evolution factors based on case studies, to be used for deciding whether to pay off TD immediately. We will elaborate on this method in Sect. 3.3.

Schmid introduces a formal mathematical method of calculating TD using highly formalized equations [12]. We will elaborate on this method in Sect. 3.4.

3 Methods

This study compares 4 different approaches for TDM. In order to properly answer our research question, we will compare these approaches on multiple criteria, taking into consideration that not all of these approaches have been formally tested or used in a realistic environment. The approaches considered in this comparison are:

- Simple Cost-Benefit Analysis
- Analytic Hierarchy Process (AHP)
- Defining Decision Factors
- Formalized Evolution

In this section we will describe these different approaches to TDM.

3.1 Simple Cost-Benefit Analysis

Seaman et al. [14] introduce the Simple Cost-Benefit Analysis, which is based on traditional financial cost-benefit analysis. It is the simplest approach to managing TD and therefore provides us with a good baseline for comparison with the other approaches.

This approach focuses on a ”technical debt list”, which contains items representing a task that is not completed (satisfactorily) and thus might cause problems in the future. Examples of this include missing tests, needed refactorings or missing documentation.

Each of these items has a **principal** and an **interest**. The principal describes how much work is needed to complete the item. The interest consists of two parts: the **probability**, which describes the chance that the debt will lead to problems if not paid, and the **amount**, which describes how much extra work the debt will cause if not paid. The probability itself depends on the timeframe for which the analysis is done. A software element that contains TD might not change in the upcoming month, but could be highly likely to change before the end of the year.

These three metrics are then assigned estimated values of high, medium or low. These are rough estimates, but sufficient in making preliminary decisions that can be worked out in more detail later. Example usage of a slightly more fine-grained approach can be seen in Fig. 1, which contains 9 god classes that should be refactored. A possible strategy for paying the debt would be to start in the upper left corner, which are high interest classes that require little effort (low principal) and then moving down and right.

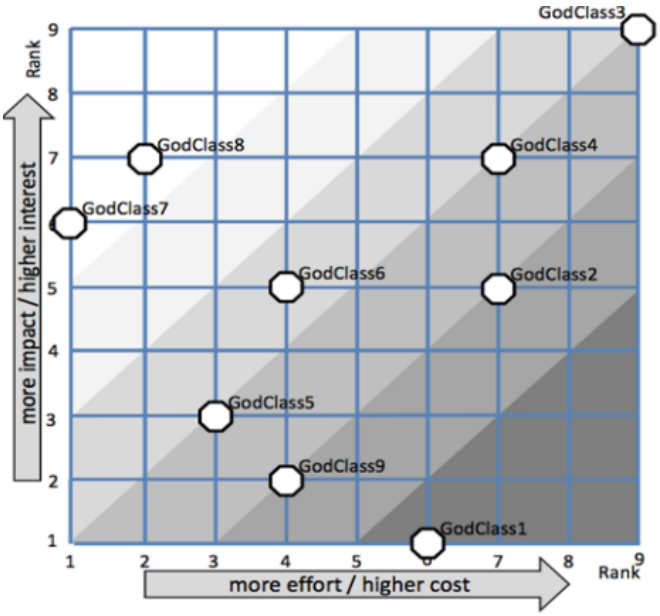


Fig. 1. Cost-Benefit matrix for nine god classes in an industrial software application. [14]

This approach increases in reliability when more data is available, especially on the impact of the TD on the rest of the application. However, even with limited amounts of data, expert opinion can be used to provide a decent approximization of the interest of an item. In addition, this approach integrates well with software metrics, making it easy to adopt into a workflow.

3.2 Analytic Hierarchy Process

Seaman et al. [14] introduce a second process to work with TD, the Analytic Hierarchy Process (AHP). This process comes from the decision sciences and is a well supported approach in literature [11].

It structures a problem solving process by comparing alternatives with regards to certain criteria, resulting in a ranking of these alternatives. It structures this process by making a hierarchy of criteria and then performing pair-wise comparisons of alternatives under these criteria, resulting in a priority vector for the alternatives.

In Fig. 2 an example is available, where the alternatives (choices) X, Y and Z are compared with respect to the criteria (factors) A, B, C and D, in order to reach a certain goal. Factors can be quantative and

qualitative which makes factors like 'product performance' possible with qualitative values like 'much better' and 'worse'. All choices are compared on all factors in order to produce a ranking of the possible choices.

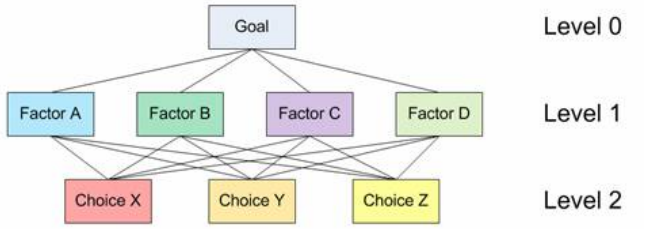


Fig. 2. Example of an AHP hierarchy. [16]

Many of the pair-wise comparisons are objective and quantitative, and can thus be automated, resulting in an approach that reduces the amount of human input required. In applying AHP to the TD problem, the alternatives would be the instances of TD in the system and the output of the process would be a ranking of these items, prioritising items that should be paid off first.

3.3 Defining Decision Factors

Snipes et al. [15] approach the TD problem from a different direction, as they do not consider the effects of the debt on related modules, but only on the module in which the debt occurs. The authors performed a survey under 7 Change Control Board (CCB) members for a product of over one million lines of source code. The CCB is responsible for deciding what changes and updates should be made in the next iteration of the product.

This has lead to the classification of six categories of TD costs.

- Investigation: The cost of diagnosing and verifying a problem.
- Modification: The cost of implementing and verifying a fix.
- Workaround: The cost of providing workarounds for a known failure.
- Customer support: The cost of providing support to customers that encounter the problem.
- Patch: The cost of providing a temporary fix to affected customers.
- Validation: The cost of testing the system in its entirity.

Not all of these costs have the same importance, as Investigation cost is estimated to be 50-70% of the cost of fixing a defect [15] and Validation cost is between 20-30%. These costs also change when a fix is deferred, as deferring a fix might lead to more customer support or patching, in addition to a workaround being required. However, Validation costs are lowered, as validating the workaround is easier than validating a fix, due to the smaller scope and impact of the workaround.

The effect of deferring on these costs can be found in Fig. 3. In this image, it can be seen that fixing a defect incurs an Investigation, Modification and Validation cost. Deferring increases the Investigation cost by having to investigate twice, due to having to investigate again when the workaround will be replaced by a proper fix. However, deferring reduces the Validation cost as the workaround is easier to test than a complete fix. In addition, Patch and Customer Support costs may be incurred if the Customer requests as such.

Cost Category	Fixing	Deferring	Condition
Investigation	→	↗	
Modification	→	→	
Workaround		→	
Customer support		→	Customer Request
Patch		→	Customer Request
Validation	→	↘	

→: Incurred, ↗: Increase, ↘: Decrease

Fig. 3. Change patterns of defect costs between fixing and deferring [15].

According to the respondents, there are multiple key factors that play a role in deciding when to fix TD. In order of importance, they are:

- Severity: The importance of capabilities affected by the defect.
- Workaround existence: Whether it is possible to implement a workaround and defer the fix.
- Urgency by customer: Has the fix been specifically requested by the customer?
- Effort to implement: Estimated effort versus resources available and the schedule.
- Risk of fix: Estimation of the extend of code and functionality will be affected and will need to be tested.
- Scope of testing: The impact of the change and the amount of testing required.

However, while these factors play an important role in TD decision-making, Snipes et al. [15] also propose a more formal way of deciding whether the TD should be paid or deferred to a later date. In order to do this, they developed a cost-benefit analysis based on the key factors identified. In the equation seen in Fig. 4, P is the cost of Investigation, Modification and Validation, representing the costs of paying of the debt right now. It therefore is the principal of the debt.

The interest is formed of multiple components. I_w represents the cost of defining a workaround and I_c the cost of Customer support. $I_{pr} * I_p$ represents the probability that a customer will request a patch, multiplied by the cost of the patch. I_{fr} then is the probability that the deferred defect will eventually be fixed, making $P * I_{fr}$ the expected cost to pay off the debt. Whenever the cost-benefit ratio $\rho < 1$ it is better to pay the principal immediately and fix the defect right now, otherwise, there is a financial advantage to deferring the fix.

$$\rho = \frac{P}{I_w + I_c + I_p * I_{pr} + P * I_{fr}}$$

Fig. 4. Equation for Cost-Benefit Analysis Ratio [15].

3.4 Formalized Evolution

Schmid [12] provides us with a highly formalized approach to reasoning about TD and its resolution. He defines TD in Definition 1.

Definition 1. $TD(S,e) = max\{CC(S,e) - CC(S',e) | S' \in Sys(S)\}$

In this formula, *TD(S,e)* is the TD for a system S together with an evolution step e, which is a single change of the system. *Sys(S)* describes all systems that are behaviorally equivalent to S, which is all systems that have the same visible behavior as S, even though the internal workings can be very different. Finally, *CC(S,e)* describes the cost of performing evolution step e on system S. This means that the

TD in Definition 1 describes the cost of performing the change on the current system S compared to the most optimal implementation of S with the same behavior.

However, evolution does not usually occur as a single change, but commonly as a sequence of evolution steps. Given an evolution sequence $\vec{e} \in E^*$, where E^* describes the set all possible evolution sequences for a system, we conclude the new Definition 2.

Definition 2. $TD(S,\vec{e}) = max\{CC(S,\vec{e}) - CC(T,\vec{e}) | T \in Sys(S)\}$

This also describes the TD as the cost of a sequence of changes relative to an optimal implementation.

Now the question remains whether it is better to pay this TD immediately or at a later moment. In order to do this, we do not just consider the planned evolution sequence ($\vec{e}_{plan}$), but also the potential evolutions ($\vec{e}_{pot}$) following it. For simplicity, we assume that refactoring or restructuring can only be done at the beginning or the end of a sequence. If c_{rest} represents the cost of restructuring, we can visualize this as in Fig. 5, where two alternatives are displayed. The second alternative can be reached by paying the costs of restructuring.

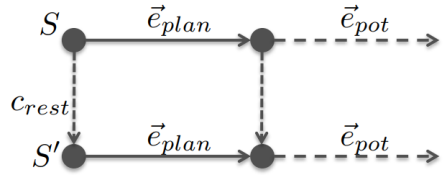


Fig. 5. Development alternatives [12].

This means that it is beneficial to refactor immediately iff

$$\begin{aligned} &CC(S,\vec{e}_{plan}) + CC(S_{plan}\vec{e}_{pot}) \\ &\geq c_{rest} + CC(S',\vec{e}_{plan}) + CC(S'_{plan}\vec{e}_{pot}) \end{aligned} \quad (1)$$

In Equation 1, S_{plan} is the state of system S after $\vec{e}_{plan}$. If the cost of implementing $\vec{e}_{plan}$ and $\vec{e}_{pot}$ is greater than the cost of refactoring and then implementing $\vec{e}_{plan}$ and $\vec{e}_{pot}$ onto this restructured system, it is beneficial to restructure.

However, this procedure has a fundamental flaw, as there is not a single path for future development, but potentially infinite different paths. In order to replace the precise values previously used, we thus introduce the notion of *expected cost* and *expected technical debt*.

For each $\vec{e} \in E^*$, where E^* describes the set all possible evolution sequences for a system, p describes a probability measure over E^*, i.e., $p(\vec{e}) \in [0...1]$, such that $\sum_{\vec{e} \in E^*} p(\vec{e}) = 1$. We use this to define the Expected Cost of a system S in Definition 3.

Definition 3. $\widehat{CC}_p(S) = \sum_{\vec{e} \in E^*} p(\vec{e}) * CC(S,\vec{e})$

The *expected technical debt* $\widehat{TD}_p$ is defined correspondingly (by using $\widehat{CC}_p$ in Definition 2). In accordance with the definition of expected TD, the calculation to determine whether it is beneficial to restructure immediately is given by:

$$\begin{aligned} &CC(S,\vec{e}_{plan}) + \widehat{CC}_{pot}(S_{plan}) \\ &\geq c_{rest} + CC(S',\vec{e}_{plan}) + \widehat{CC}_{pot}(S'_{plan}) \end{aligned} \quad (2)$$

S' is the state after performing the restructuring corresponding with c_{rest}. Equation 2 shows that the *estimated technical debt* is always impacted by the uncertainty of development, which means that we can never predict the optimal way of handling TD.

3.4.1 Approximating formal optimization

The analysis of formal evolution so far relies on a number of optimizations that can lead to an infinite search space for identifying and handling TD. Three problems are encountered that can be solved using an approximation:

- The number of behaviorally equivalent systems can be very large, potentially infinite, which makes finding an optimal system, used in Definition 1 and Definition 2, highly time consuming. This problem will be addressed by introducing a fixed relative system.
- The number of potential evolution sequences grows exponentially with the length of those sequences. Under certain circumstances we can restrict our analysis to single evolution steps (see Definition 8).
- After applying the previous two optimizations, the number of potential evolution steps might still be infinite or too large to consider properly. Thus we look at whether it can be replaced with a finite set and what the consequences are of doing so.

In order to select a fixed relative system, we need a system that approximates an optimal solution and is good enough to fill the role of optimal solution. Often, an expert solution is available that can function as an optimal solution S'. We use this to define *relative technical debt* RTD as per Definition 4.

Definition 4. $RTD(S,S',\vec{e}) = CC(S,\vec{e}) - CC(S',\vec{e})$

Expected relative technical debt $\widehat{RTD}$ can be defined correspondingly using the *expected cost* $\widehat{CC}$, which can again be used to calculate whether in it beneficial to restructure right now.

Using an expert solution instead of a theoretical optimal solution brings this method a step closer from theory to practice. However, the CC function still requires exhaustive probability calculation over an infinite number of evolution steps and sequences. To deal with this, we replace the potentially infinite space of sequences with a representative finite subset.

Definition 5. $c_{rest} \leq \sum_{e \lessdot \vec{e}_{plan}} RTD(S,S',e) + \widetilde{RTD}_n(\vec{e}_{plan}(S),\vec{e}_{plan}(S'))$

Definition 5 allows for approximating c_{rest} (the restructuring cost of an evolution plan). It consists out of two components. First, the sum of the relative technical debt (RTD, Definition 4) for each evolution step e. That is, the TD caused by only the specific step onto the system.

Second, the approximated relative debt of the sequence as a whole. The idea is that evolution steps implemented together may do even more harm than implementing them separately (e.g. a harmful dependency between the two). This type of relative debt is hard to define, but it can be approximated using the following definitions:

Definition 6. $\tilde{p}(e) = \begin{cases} \sum_{\vec{e} \in E^*, e \lessdot \vec{e}_{plan}} p(\vec{e}) & e \in \tilde{E} \\ 0 & e \in E \setminus \tilde{E} \end{cases}$

Definition 7. $\widetilde{STD}_{\tilde{E}(S)} = \sum_{e \in \tilde{E}} \tilde{p}(e) * TD(S,e)$

Where earlier definitions for (R)TD are correct, they are not practically usable due to their infinite nature. In Definition 6 and Definition 7 E^* is the set of all potential evolution sequences (infinite). E is the set of all potential evolution steps (infinite) and $\tilde{E}$ is a finite representative subset of E, containing n sequences. This set consists only of the concrete evolution steps a business is considering and is thereby finite. Definition 6 describes a probability function for evolution paths consisting of only steps in $\tilde{E}$. Definition 7 describes the approximated technical debt given the weighted potential evolution paths. This is critical since it allows for estimation of the complex interdependent

TD of evolution steps using a finite set of possible evolution sequences. $\widetilde{RTD_n}$ can be derived from Definition 7 and Definition 4.

Although Definition 5 is usable through the use of only a finite set of potential evolution paths, the formula is still computationally complex. Schmid [12] proposes that the debt coming from the interdependence of evolution is negligible. This allows for simplication of Definition 5 into the following:

Definition 8. $c_{rest} \leq \sum_{e \in E'} RTD(S, S', e) * p'(e)$

$E' = \tilde{E} \cup \{e | e \in e_{plan}\}$ and p'(e) is identical to p(e) except that it returns probability 1 for all steps e already part of the evolution plan e_{plan}.

This final definition allows for relatively simple decision making since it gives a clear definition of the TD that can be compared to the restructuring cost. It sums the RTD of all evolution steps part of the plan and the RTD of all other potential evolution steps, weighted by their likelihood of occurance. This concludes the formal method proposed by Schmid.

4 Results

We now go over the different methods that are described in Sect. 3 by comparing them on several properties.

- Accuracy: how accurately a method predicts the TD state of an item.
- Automation: in what capabilities can the method be automated.
- Completeness: the capability of a method to consider all relevant factors when quantifying a TD item.
- Feasibility: how likely it is that companies will adopt a method and/or how easy it is to integrate in the workflow.
- Flexibility: how easy it is to change the method to suit the needs of the company.
- Transparancy: how clear it is to the users how the result has been reached.
- Workload: the effort required to use the method.

This list of properties is non-exhaustive and the properties have been chosen based on how well they cover the entire process of decision-making, in order to look not only at technical aspects such as completeness and accuracy, but also from more user-related perspectives such as workload and transparency. In addition, they have been selected based on whether they could be derived from the discussed articles.

Table 1 summarizes our findings. Every method is given a score for every property that ranges from 1 (very low) to 5 (very high).

Table 1. Overview of methods and their scores. Methods are abbreviated as follows: Simple Cost-Benefit Analysis (SCBA), Analytic Hierarchy Process (AHP), Defining Decision Factors (DDF) and Formalized Evolution (FE).

	SCBA	AHP	DDF	FE
Accuracy	1	3	3	5
Automation	2	4	2	3
Completeness	1	4	2	5
Feasibility	3	4	3	2
Flexibility	2	5	3	3
Transparency	5	2	4	2
Workload	5	3	3	1
Total	19	25	20	21

4.1 Simple Cost-Benefit Analysis (SCBA)

What makes Simple Cost-Benefit Analysis stand out is its simplicity. It has only three components (principal, interest amount and interest probability) which makes it attractive to add to a workflow. Therefore, SCBA scores high on feasibility, workload and transparency.

However, the simplicity is also its weakness. The equation used by SCBA does not consider factors beside principal cost and interest in the cost of developing new features. Take for example the factor 'performance'. If incurring TD decreases the performance of a system by 10 percent, this will only affect end users of the system but will not generate a growing interest of development cost. However, it might lead to losing customers and thus a different form of interest. Completeness and accuracy scores of this method are therefore low.

4.2 Analytic Hierarchy Process (AHP)

The strength of AHP lies in its flexibility and its pragmatism. AHP sets no constraints to the factors that are used, making it flexible and also potentially covering all relevant factors. Furthermore, it breaks down the question of factor prioritization into small comparisons: each factor is directly compared to other factors and the algorithm generates a general prioritization from that. Besides setting up the matrix for comparing features, most of the operations can be automated.

The disadvantages of AHP are not obvious, yet they are present. The flexibility brings great potential but it requires the user of the method to know how to configure it. Other methods come with their own balancing based on research which (although less flexible) might yield better results. The mathematics underlying the method also makes it less transparent to some users.

4.3 Defining Decision Factors (DDF)

DDF does not stand out positively nor negatively. It achieves some accuracy and completeness by considering all steps part of fixing or postponing an issue. However, its equation (Fig. 4) is too simplistic, as it does not consider the effects of time on factors such as I_c or I_{pr}, nor the opportunity cost of the other tasks that could be done instead. This method also suffers from the fact that the effect on the rest of the system is left out of the equation.

The input on DDF then also requires broad knowledge of the system. Factors like 'urgency by customer' and 'risk to fix' are best answered by different people in an organization. This makes the method more complex to adopt.

Finally, the method is very clear. This increases the transparency of it and also has a positive effect on feasibility and (partial) automation.

4.4 Formalized Evolution (FE)

This approach provides some significant benefits, but also has its fair share of drawbacks. The formalization offers a precise analysis of TD, which also shows that we require an infinite search space and need to identify theoretically optimal systems, which is not feasible in practice. Approximation can alleviate these issues, but also causes deviation from the 'real' values. The major drawback of this method is the workload it brings. It requires to predict many evolution paths to calculate the restructuring cost. It is possible to use very few, but this directly impacts the accuracy of the method.

Furthermore this method is built on one abstract cost function (CC). This function is flexible in the sense that no hard definition of it is given besides: *"the (minimal) cost required to perform an evolution step on a system"*. If we assume that this only entails the cost to implement an evolution step, it is both transparent and relatively easy to use, since cost is something businesses are already familiar with. However, it also means that similar to SCBA and DDF some factors might be left out of the equation.

Automation of this method is doable as it is mostly a mathematical model that directly allows comparison of alternatives. However, the implementation costs are still required as an input and might be hard to calculate depending on how many factors are included in it.

In general, this method provides strong theoretical foundations for further decision-making approaches and research, while also being usable in realistic environment. Although the workload to use it is high

and the algorithm is not easy to understand, it can yield very accurate results.

5 Discussion

An important note to make is that although most relevant research has underlined the need for refactoring, it is not imperative that investing time in decision making is beneficial.

Definition 9. $T_{saved} = T_{td} - T_{dm}$

In Definition 9 T_{td} represents the time saved by refactoring, T_{dm} the time taken for decision-making and finally T_{saved} is the net time saved for the company.

This formula makes clear that although a decision-making method can be very good in finding the right moment to fix TD (thereby saving time in the future) it needs to be proportional to the TD it proposes to fix. Otherwise, the process as a whole is still not beneficial. It also shows that improving the decision-making process ad infinitum is not always beneficial as it might increase the workload so much that the total T_{saved} becomes negative.

In this paper, we considered four different approaches to technical debt management. We compared these approaches based on a wide range of factors, both technical and organisational. This offers an overview that is relevant to varying groups of users. However, there are always more factors that can be taken into account and doing so might give a different result. Furthermore this overview suffers from the problem that several of these approaches are untested in industry or case studies. This makes their results unreliable at best, as theoretical results do not guarantee the same results in practice. Therefore, the results of this paper should be considered a comparison of theoretical results.

6 Conclusion

Technical debt (TD) is becoming a major issue in software development and a factor in its decision making. Therefore, methods are required to deal with TD that help with finding the right moment to resolve it. Research has been conducted on the topic of TD already, providing a variety of methods to deal with it. These methods typically solve two problems: how do we quantify TD and how do we compare candidate solutions for resolving TD?

We discussed four methods that attempt to answer these questions. Simple Cost-Benefit Analysis (SCBA) is a simple model that assigns principal and interest to technical items, similar to models found in finance. Defining Decision Factors (DDF) categorizes the cost of TD into six defining factors which are used to choose between incurring and resolving TD. Analytic Hierarchy Process (AHP) provides a more general tactic that is grounded in decision-making literature. Users can define all factors they deem relevant. AHP orders them and is able to rank technical item importance based on feature importance and scores given to the technical items. Finally, Formalized Evolution (FE) builds a mathematical and statistical model based on the cost of implementing an item and the debt it brings. It provides an accurate prediction of whether a refactoring will pay off, given the correct item costs and potential evolution paths.

All four methods have their benefits and disadvantages. SCBA is easy to use but not complete. DDF provides good insight in the cost of TD, but requires precise input and lacks non-technical factors. AHP is flexible and well automatable. However, it requires so much input that the approach itself does not provide much more than a framework for decision making. FE is very accurate but more complex in use and very time consuming, although some parts can be automated. By comparing and scoring the methods on several criteria, we found that AHP came out on top as the suggested approach to use.

Answering the question of whether and when to resolve TD based on these approached is not yet feasible without any case studies. Choosing a most feasible approach suffers from this same problem, as theoretically being the most appropriate approach does not guarantee that it will be so in practice. Although all these methods are very promising, more research is needed in the form of case studies that apply these methods in practice. Furthermore, the feasibility of the decision-making process itself should be subject to research especially when its complexity increases.

Acknowledgments

The authors wish to thank our reviewers and J. Tan as our domain expert.

References

[1] W. N. Behutiye, P. Rodrguez, M. Oivo, and A. Tosun. Analyzing the concept of technical debt in the context of agile software development: A systematic literature review. *Information and Software Technology*, 82:139 – 158, 2017. doi: 10.1016/j.infsof.2016.10.004

[2] W. Cunningham. The wycash portfolio management system, 1992.

[3] B. Curtis, J. Sappidi, and A. Szynkarski. Estimating the size, cost, and types of technical debt. In *Proceedings of the Third International Workshop on Managing Technical Debt*, MTD '12, pp. 49–53. IEEE Press, Piscataway, NJ, USA, 2012.

[4] T. T. Ho and G. Ruhe. When-to-release decisions in consideration of technical debt. In *2014 Sixth International Workshop on Managing Technical Debt*, pp. 31–34, Sep. 2014. doi: 10.1109/MTD.2014.10

[5] J. Jeremiah. Survey: Is agile the new norm?, 2015.

[6] D. Leffingwell. *Scaling software agility: best practices for large enterprises*. Pearson Education, 2007.

[7] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *The Journal of Systems & Software*, 101:193–220, 2015.

[8] A. Martini and J. Bosch. Towards prioritizing architecture technical debt: Information needs of architects and product owners. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, pp. 422–429, Aug 2015. doi: 10.1109/SEAA.2015.78

[9] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. In search of a metric for managing architectural technical debt. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pp. 91–100, Aug 2012. doi: 10.1109/WICSA-ECSA.212.17

[10] A. Nugroho, J. Visser, and T. Kuipers. An empirical model of technical debt and interest. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, pp. 1–8. ACM, New York, NY, USA, 2011. doi: 10.1145/1985362.1985364

[11] T. L. Saaty. *Decision making for leaders: The analytical hierarchy process for decision in a complex world*. Lifetime Learning Publications, 1982.

[12] K. Schmid. A formal approach to technical debt decision making. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13, pp. 153–162. ACM, New York, NY, USA, 2013. doi: 10.1145/2465478.2465492

[13] C. Seaman and Y. Guo. Chapter 2 - measuring and monitoring technical debt. vol. 82 of *Advances in Computers*, pp. 25 – 46. Elsevier, 2011. doi: 10.1016/B978-0-12-385512-1.00002-5

[14] C. Seaman, Y. Guo, N. Zazworka, F. Shull, C. Izurieta, Y. Cai, and A. Vetr. Using technical debt data in decision making: Potential decision approaches. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pp. 45–48. Zurich, June 2012. doi: 10.1109/MTD.2012.6225999

[15] W. Snipes, B. Robinson, Y. Guo, and C. Seaman. Defining the decision factors for managing defects: A technical debt perspective. In *2012 Third International Workshop on Managing Technical Debt (MTD)*, pp. 54–60. Zurich, June 2012. doi: 10.1109/MTD.2012.6226001

[16] K. Teknomo. Analytic hierarchy process (ahp) tutorial. `https://people.revoledu.com/kardi/tutorial/AHP/AHP-Example.htm`, 2006. Accessed: 2019-03-11.

[17] E. Tom, A. Aurum, and R. Vidgen. An exploration of technical debt. *J. Syst. Softw.*, 86(6):1498–1516, Jun 2013. doi: 10.1016/j.jss.2012.12.052

[18] H. Van Vliet. *Software engineering: principles and practice*, vol. 13. John Wiley & Sons, 2008.

An Overview of Technical Debt and Different Methods Used for its Analysis

Anamitra Majumdar and Abhishek Patil

Abstract— Technical debt is a widely used term in software development which is the cost of restructuring the code as a result of flaws present in the software system. This is caused due to focusing on short-term benefits rather than thinking about the long-term life of the software. Most of the time, developers do not worry too much about the overall health of the software during development and use low quality code in the process to meet their goals quicker. This results in code smells, bugs, performance issues, security loopholes and unreadable code. Ignoring these problems is the same as going into debt i.e., choosing to ignore the problems so as to "borrow" time and push out releases quicker. Many instances of debts incurred are also unintentional like in cases of updates or patches to a software and some can even be necessary when, for exmaple, a deadline needs to be met. These debts pressurize the developers to revisit the same code in order to work on it again. Analysis and quantification of technical debt is necessary as it gives an idea to the developers and the stakeholders of the time and resources required to manage the debt. It also gives us an overall perspective as to why one should be careful while going into debt and what can be done to alleviate the problem. In this paper, we discuss in detail about what technical debt actually is, its types, how it impacts the software lifecycle in the long run and how developers work on paying back the accumulated debt. We then review the approaches given in four research papers to measure different types of technical debt in open source software projects and study their effects on maintainability and debt payback. In the end, we conclude by summarizing our paper and put forth our own idea on how to reduce the problem of technical debt in the development stage.

Index Terms—Technical debt, restructuring, developer, software, analysis

1 INTRODUCTION

Technical debt is the cost of the extra work needed to reorganize the code as a result of the shortcuts or loopholes used by developers to get to meet their short-term objectives quicker. Often the debt that builds up is due to the constant updates to the software as opposed to the fault of the developers themselves.

Ward Cunningham was one of the first people to coin this term and it has been popular ever since. In his words: "*Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with refactoring. The danger occurs when the debt is not repaid. Every minute spent on code that is not quite right for the programming task of the moment counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unfactored implementation, object-oriented or otherwise.*"[1].

Most teams around the world do not bother much about technical debt as they are focused purely on the end goal rather than developing a good quality software. This is a problem as someone at some point has to go back to the code anyway due to the debt that has been accumulated so it is best to take care of the problem right at the initial phase. We discuss about this at the end of the paper in the conclusion section.

In this paper, the concept of technical debt is explained in detail, its types are explored and we also see how it affects the general software code. We then summarize the different approaches used by researchers in the past to measure technical debt in open source software projects and compare them in terms of efficiency and maintainability.

The rest of the review paper is organized as follows: section 2 introduces the concept of technical debt, how to identify it in a software and the most common types of debt that we can see in a software. Section 3 explains in detail the impact of technical debt on software systems, the people working on the software and the users of the software. Section 4 discusses how to manage technical debt or in simpler terms, how developers work on paying back the debt so as to mitigate it from the software. Section 5 discusses the different approaches that have been used to quantify technical debt in the past. Here, we review the techniques from four different research papers. Section 6 highlights the threats to the validity of the findings of the reviewed papers. Lastly, section 7 gives a brief summary of the contents of this paper and concludes with our own theory on how to reduce technical debt.

2 THE IDEA OF TECHNICAL DEBT

The Software Engineering Institute at Carnegie Mellon University says that technical debt "*conceptualizes the tradeoff between the short-term benefit of rapid delivery and long-term value*"[2]. Suppose you are a software developer and have to add some new feature to the already existing code. There are two ways this can be achieved:

i) Use not up-to-par code or messy code, which is a lot easier.
ii) Use properly structured code without any loopholes, which takes time.

Technical debt is very similar to a financial debt. If a code keeps incurring debt, the amount of work that the developer would need to put in later will be much more, which is similar to the higher interest paid over time in terms of financial debt. The longer the debt is ignored, the more will be the software entropy which in turn will result in complications.

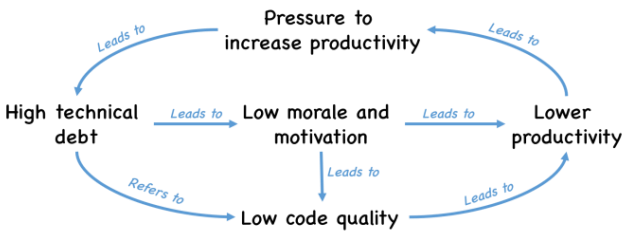


Fig. 1. The cycle of technical debt[3]

Technical debt does not include delaying or not adding certain functionalities to a software. It solely consists of bad code and/or

bugs that have been caused as a result of not using proper code. Technical debt can be used to refer to the debt in any stage in the entire software development cycle but the term is generally used just for the programming phase.

2.1 Identifying Technical Debt

There can be a lot of ways to detect technical debt once the code is analysed, including:

- Code smells, which indicates problems on the surface that leads to potentially deeper problems in the code
- High complexity, when the technologies used cannot interact with each other in the right way
- Bugs in the software
- Improper style of coding, which is generally fixed by following guidelines
- Increased load times, which is an indicator that the code may not have been optimized correctly
- Performance issues, which arise due to incorrect memory allocation
- Software age, which correlates to usage of outdated libraries, tools and technologies

Most developers can identify debt in a software right away but there are some cases when even the savviest programmers have a hard time analysing the code and finding instances of debt.

2.2 Types of Technical Debt

There are four basic situations which can result in the accumulation of technical debt.

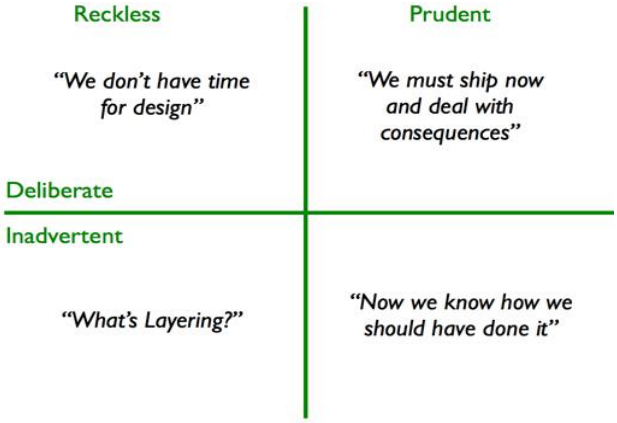


Fig. 2. The four quadrants of technical debt[4]

Each situation has its own importance and value:

- Reckless/Deliberate Debt: where developers are only focused on the end goal
- Prudent/Deliberate: where developers are aware that they are in debt but choose to deliver now and fix later
- Reckless/Inadvertent: where developers lack proper expertise to identify debt
- Prudent/Inadvertent: where developers realize that the software has debt once they run the whole code

There are three main types of technical debt:

i) Strategic technical debt

This occurs when the developers make a well-informed decision to induce debt in the code knowing fully well the potential risks it might have later on. These kinds of decision are most often collective and the entire organization has a role to play in it. For this reason, this is also known as planned technical debt. It is very important as thorough as possible while making such compromises and planning out the entire timeline right from the moment a shortcut is used to actually paying it back once the debt has accumulated. An example could be: skipping code reviews and unit tests so as to meet an immediate deadline. In these cases, it is critical to maintain a record of all the debts so that it is easier address them in the future.

ii) Unintentional technical debt

This type of debt is not planned in advance and arises due to poor coding. This happens mainly due to the fault of the developers working on the project or software. Many people do not care too much about the code quality and keep pushing new releases without taking care of the issues within the code. It is common when the best possible solution is not followed due to not being aware of the specific guidelines or the design of the modules. There are also times when the goals of the overlooking board or the policy makers do not align with that of the developers and this results in mismatch and confusion on both sides. Without proper communication, this kind of debt is bound to occur. This can also be called naïve technical debt.

iii) Unavoidable technical debt

This is the kind of debt which occurs when there is a new technology which makes the old code obsolete or when the goals change mid-project as a result of which new features or functionalities have to be added in. There may also be debt due to the patches made to a fully released software. The patches can be very minor such as security fixes or heavy such as UI overhaul or design changes. For example, to modify a website so that it supports better mobile viewing, the entire design unit has to be re-coded which increases debt. Lastly, this can result even due to the lack of expertise on the developers' part.

2.3 Reasons for Technical Debt

There can be both good and bad reasons for technical debt. In any case, it is imperative to keep track of what is going on so that the team working in the project do not get lost or slowed down. If the team gets affected, so do the future releases of the product.

Technical debt can be good in a situation where the release of the product is more important than the internal structure or design of the code. If the end user cares only about the working of the product and not it's quality, the focus of the organization will obviously be on making a stable release as quick as possible. This will not only satisfy the customer but also be wise on the company's part as they save time and deliver on their promises. However, if the requested product demands that level of design, the developers will have to spend more time on cleaning up the code so that it is not messy in the final version.

An example of a bad reason for technical debt is when the team working on a project is more focused on other areas because they feel like they can work better on them or find it more interesting. Situations often arise when developers who have been working on a piece of code for a long time get frustrated due to not reaching their targets and that is when they tend to cut corners to push their complete code sample.

Even if there is a very good reason for inducing debt in a software for all the right reasons, simply choosing messy code just to release a complete software is not the end of the story. The developers working on the software must come back to the code at some later point to reorganize the code so as to eliminate all instances of debt. Otherwise it will keep getting buried with all the extra coding on top of it and eventually create problems in the functionality of the product. Any team developing a product must consist of both types of developers i.e., a programmer who is quick in reaching the defined targets and a programmer who takes time and analyses the code as it is being developed ensuring that no shortcuts are used in the process.

3 IMPACT OF TECHNICAL DEBT

The effect that technical debt has depends on a lot of factors such as the premise with which the debt was incurred, how much debt has been accumulated, the expertise of the developers, the type of debt, quality of the existing codebase and much more. There are several ways in which technical debt can affect the software, environment and end users:

i) Slower productivity

As the debt rises, so will the complications of adding new features to an existing codebase. This happens due to the fact that the code has become obsolete and/or difficult to work with as result of poor design. This in turn creates problems for the end user as he has to wait longer for the final version of the software.

ii) Longer release cycles

With the rising amount of debt, the time needed to reorganize the code will also increase and the release dates get pushed back further. This happens due to the huge changes that need be made to the code. What should ideally take a day can possibly take more than a week.

iii) Slower build times

When dealing with a software with a high number of lines of code (LOC) i.e., large projects, the time to build the overall module is a necessary factor that should not be forgotten about. If the codebase keeps incurring debt, this will impact the build time in a negative way due to the improper structure and bugs present inside.

iv) Difficult to add new features

The developers have a hard time adding new functionalities to the existing code because of its complex nature. For example, if the database being used in a software is not properly organized, working with it becomes difficult.

v) Testing becomes harder

If the code is riddled with bugs due to accumulated debt, the testing will be more complex. For testing to be carried out, one needs a well-balanced system with a smooth flowing functionality between the different modules of the software. Cases of debt hinder this harmony and as a result, testing becomes a painstaking process.

vi) No stability

A software can even crash or some of its component might fail due to the bugs present inside or certain optimization issues. This is particularly dangerous in situations where end users are dealing with a lot of data and demand perfect uptimes. It also contributes towards shortening the overall life of a software.

vii) Mindset of developers

As the instances of debt go up, the developers working on the project get frustrated and gradually lose interest. This also affects new people wanting to join the venture as no one wants to work on a sub-par software system. This carries on and becomes a bigger issue as long as debt is ignored.

viii) Financial burden

Going into debt also has several financial impacts. Restructuring the code to pay back the accumulated debt not only requires skill but also resources which have a cost. This is one of the primary reasons why organizations have to be careful while developing softwares. Even a hint of carelessness while developing could cost the organization a lot of money to fix the resulting issues.

ix) Cumbersome updating

If the code is a mess or has smells, updating it will be quite difficult considering the fact that developers would need to revisit entire portions of the code just to find out what went wrong and how they can fix it. This delays the updating process of a software unnecessarily. Sometimes the software cannot function at all unless it has been updated.

x) Maintenance becomes tough

It is difficult for a developer to work with low quality code. Even basic maintenance of a software system like security patches or layout changes take time when the code structure is not right. If a software system is not maintained regularly, it will create problems in itself.

xi) Hardware replacement

There are times when developers working on a project use new hardware to incorporate better functionality in the system or use it as it is a necessity for the features they want to add. This can create compatibility issues with the code itself.

xii) Dissatisfaction for end users

One cannot expect the end user to be content with a software if it has issues due to technical debt. This not only annoys the customer but also damages the reputation of the organization making the software. As a result of the bugs, frequent patching is needed which is once again not pleasant for the customer.

xiii) Excessive load on helpdesk

The customer services team also have a hard time catering to all the issues faced by the end users. A lot of their time is used up in addressing the problems caused due to debt. Though their job is to help customers in any case but if there is no debt, they have time to focus on other issues too.

4 MANAGING TECHNICAL DEBT

Managing technical debt is a complex process involving many iterations over the lifecycle of a software. The more the debt is ignored, the longer it will take for developers to pay back. Therefore, it is always advisable to manage technical debt as soon as possible [5].

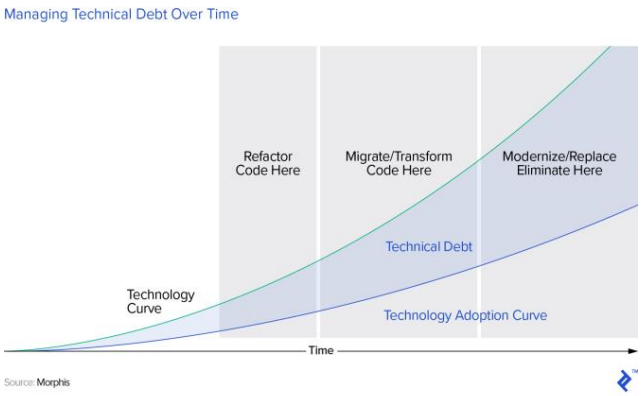


Fig. 3. Technical Debt v/s Time

Broadly speaking, technical debt follows an upwards curve while going through the stages of refactoring the code, transforming the code and eliminating or modernizing the software. For the same stages, the technology adoption curve i.e., acceptance of the software also follows a similar curve, though it has a slightly lower gradient.

There are several phases involved in properly managing technical debt, which are discussed below.

4.1 Assessment

Assessment is usually done once there are enough identifiers that lead towards evidence of debt in the code. Technical debt has to be

measured in a way such that it helps the assessment process. One way to do this is by figuring out how much time a developer needs to put in to resolve the issues present in the system. Some issues are fixed in seconds such as sections of code being accidentally commented out but others are more complicated to work with such as control flow statements being nested too deeply. The time taken to refactor a code is directly proportional to the amount of resources to be used. This provides an excellent time vs. cost assessment which ultimately helps the organization. In addition to this, it is also necessary to predict how the technical debt will change in each stage of the software lifecycle.

4.2 Informing

The next important step is letting the stakeholders know the true extent and cost of the debt. This is especially imperative when dealing with softwares that have a lot of non-IT elements related to it. In other words, proper communication is the key to managing debt as it is crucial that both the parties (authority and developers) know what is going on. Most developers cannot explain what the issue is in non-technical terms so it is necessary to have a person that bridges this gap between the developers and the stakeholders.

4.3 Implementation

Once the complete assessment of the software is done and the issues have been communicated properly, it is time to eliminate technical debt. This can be done in three main ways:

i) Get rid of the requirement

This is the simplest solution possible out of all. An organization can make a conscious decision to not change anything in the code and keep the software system in its current state. These decisions are generally made after much thought about what repercussions it might have and if the issues affect the overall functionality of the system.

ii) Refactoring the code

This involves restructuring the entire codebase where issues are present and update it such that it does not affect the functionality or behaviour of the system. This is perhaps the most common solution and developers have to spend a lot of time while refactoring as they have to carefully analyze each line of code.

iii) Replace the software

Once can always replace the entire software with a newly designed one. While this does seem fat fetched, it is practiced frequently in organizations where time is a limitation. The new application might have technical debt of its own but this solution is useful in minimizing large amounts of built up debt.

iv) Be aware of debt during development phase

If the developers are conscious about avoiding debt right from the beginning, it can make a huge difference. This is basically common knowledge but since most of the developers do not worry about going into debt at all, this is very important. If the mentality of maintaining the proper quality of code can be incorporated from the start then the instances of debt go down.

v) Refactor at regular short intervals

The practice of restructuring the code samples as soon as they are written or when a new module has been developed greatly reduces the need of revisiting the code later on. If this is done, debt doesn't accumulate as much.

vi) Put quality guidelines in place

Quality guidelines are an excellent way to remind the developers to maintain the quality of code at each step of the development process. These guidelines should be strict, well documented and followed religiously. This is similar to the guidelines that a product manufacturer has to go through i.e., ensuring that the final product is stable and debt-free.

vii) Testing and code review

Testing is an integral part of the development process of a software. It enables a lot of issues and bugs to be addressed quickly, which in turn reduces debt. Reviewing code at every major point is also just as useful and the developers get a concrete idea of how a particular code snippet could potentially incur debt.

viii) Decrease the number of dependencies

Most softwares make use of a large number of dependencies and the libraries that go along with them. Although having no dependencies is not possible but efforts should be made to keep them to a minimum. Code can become messy and incompatible if the proper dependencies are not found during runtime. If a library is added to the project instead of referencing to an external source, compatibility issues are alleviated.

ix) Using open source softwares

Since open source softwares do not have licenses attached to them, anyone can use it as per their wish. Hence, the funds for acquiring the licenses can now be used to better the code by refactoring. This is more of an indirect approach but can be a great way of reducing technical debt.

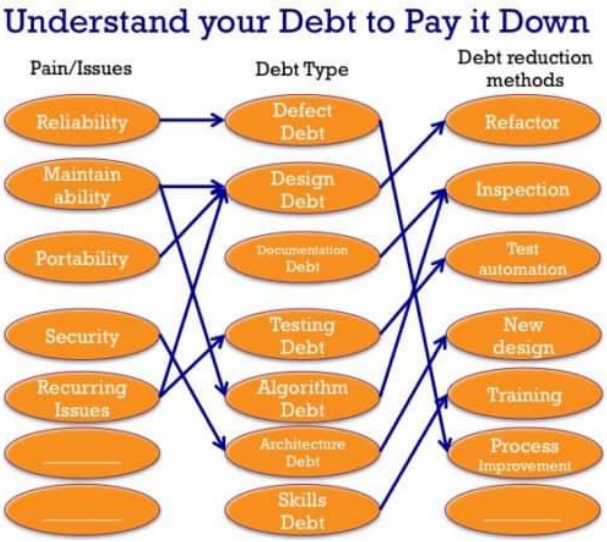


Fig. 4. The structure of paying back debt[6]

5 DISCUSSION

In recent years discussions regarding technical debt have dominated research in the field of computer science. There have been numerous studies in the past on different techniques to quantify technical debt in a system. Some of these studies have been discussed in the following sections.

5.1 Diffuseness of code smells and its impact on maintainability

Fabio, Gabriele and Massimiliano (2017) conducted a study of 30 open source systems across 395 releases[7]. Thirteen different kind of code smells were considered for research on the diffuseness and its impact on maintainability (mainly change proneness and fault proneness). Their research indicates that:

- Long method smells and complex class smells which are characterised by complex code are highly diffused. On the other hand, feature envy smell, lazy class smell and message chain smell is poorly diffused.
- Classes exhibit significantly higher change and fault

proneness when they are affected by code smells. It was also observed that higher the number of code smells, higher is the change and fault proneness.

Although class change proneness can benefit from code smell removal, code smell presence is not necessarily the direct cause of class fault proneness. Both of them are rather co-occurring phenomenon.

5.2 Self-admitted technical debt

Self-admitted technical debt (SATD) refers to technical debt instances intentionally introduced by the programmer like temporary defect fixes which are explicitly documented in code comments. Gabriele and Barbara conducted a study on 159 active projects belonging to mainly two software ecosystems Apache and Eclipse[8]. Their study states that:

- Most diffused type of SATD is code debt (30% of all analysed instances) followed by defect debt (20%) requirement debt (20%) and design debt (15%).
- SATD instances increase with introduction of partly fixed patches and stays in the system for over 1000 commits. In majority of the cases the developer repaying the debt is the same who introduced it if not then it is usually a developer with a higher experience than the one who introduced it.

From the results, no correlation was found between code files internal quality and self-admitted technical debt instances.

5.3 Evolution of technical debt in Apache ecosystem

Georgios, Mircea, Alexander and Paris (2017) analysed 66 Java open source projects from Apache ecosystem focussing on evolution of technical debt over last five years[9]. Their findings suggest:

- There is no monotonic upward trend in evolution of technical debt for the analysed open source projects.
- Technical debt along with source code metrics increase in majority of systems. However, normalized technical debt decreases over a period of time. According to them this might be due to evolution of system over a period of time or developers focus on elimination of technical debt during software lifecycle but there is no concrete evidence to support this reasoning.
- During development phase top ten most frequent rule violations account for more than 40% issues in the system.
- Technical debt due to code duplication is the most expensive to fix, estimation of this effort depends on the cardinality of the clone. Technical debt due to exception handling is also expensive for remediation.

5.4 How technical debt is paid back by fixing issues

Georgios, Mircea, Alexander Apostolos, and Paris (2017) carried out a case study on 57 Java open source projects from Apache ecosystem focusing on the amount of technical debt paid back and the types of issues that were fixed in mean time[10]. High level observations from their case study are:

- The highest fixing rate is present in projects nutch and jmeter in which more than 70% of the issues that appeared during their evolution have been fixed. This could due to the fact that the development team of both these projects use SonarQube, a static code analysis software, to evaluate the quality of internal code.
- There is no relation between absolute number of issues found and percentage of fixes. They also found that there was no correlation between size of project and defect fixing rate.
- Only a small portion of issue types account for majority of the issues in the system and these most frequently occurring issues are the easiest to fix.
- Most frequently fixed issues do not necessarily account for highest amount of technical debt paid back. Technical debt paid back might be considerable due to code duplication, this is because the estimation of the payback effort is proportional to the number of duplicate blocks of code.

Their results show that nearly 20% of issues are fixed within one month from its introduction in the system, with 50% are fixed within one year and smaller number of defects which might not be of importance to the developers are not fixed even after 10 years.

6 THREATS TO VALIDITY

There can be the following general threats to the validity of the acquired results in the reviewed papers:

- Since the analysis has been carried out for a fixed number of open source projects, it is therefore not possible to generalize the results that have been found for all the projects in existence.
- The calculations done in each paper might be imprecise as they are dependent on external factors like the effectiveness of SonarQube for analysis of code and mining the instances of SATD from the comment lines.
- Since the research questions in the reviewed papers are investigated through a case study, the reliability of the sources can be questioned. If the past experiments are not carried out in the same setup and environment as it was originally done, different results might be yielded.
- Experience of the developers who worked on the projects is not taken into account. A developer who has less experience and has worked on a large project with many others won't go into debt that much due to the combined expertise of the team and vice versa.
- The open source projects have been analysed over a specific period of time. There might be some or no debt in that period which affects the results greatly.
- Manual or operational errors by the researchers can also lead to flaws and imprecise calculations.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have explained the whole notion around what technical debt actually is. As we have seen, there are various identifiers which hint at instances of technical debt. There are different types of technical debt possible based on whether it is deliberate or inadvertent. There can also be both good and bad reasons for incurring debt.

We have also seen that technical debt can have a large number of impacts on the software itself and the environment. These impacts have mixed complexities according to their types.

Managing technical debt is a challenge and generally involves assessing the code first, followed by communicating the results to the appropriate stakeholders and then implementing ways to reduce the accumulated debt. This is crucial in handling the debts.

Lastly, we have summarized the various approaches used in the past by people in their research to quantify technical debt and answer the corresponding posed research questions. In the first paper (sec. 5.1), although the analysis of the projects indicates that some types of smells are more frequent than others, it can be safely concluded that code smells are directly proportional to the change and fault proneness of a system. The second paper (sec. 5.2) measures the diffuseness of self-admitted technical debt (SATD) in a number of active projects. The most common type of SATD is code debt and the least common is design debt. The paper also goes on to prove that instances of SATD increase with the change history of a project. Finally, there was no relation found between the actual code quality (based on attributes of coupling, complexity and readability) and the

number of SATD instances introduced. The third paper (sec. 5.3) we reviewed looks at the evolution of technical debt over the version history of a number of selected projects from the Apace ecosystem. It has been concluded based on the analysis of these projects that there is an upwards trend of technical debt with time but the normalized technical debt has a rather interesting downwards trend. It was also observed that rule violations are the most frequent types of debt. Also, code duplication is the costliest to fix in terms of time and resources used. The final paper (5.4) analyses how much debt is paid back or in other words, the issue fixing rate. No apparent relation was found between the issue fixing rate and number of issues present inside the system or between the most commonly fixed issues and total debt paid back.

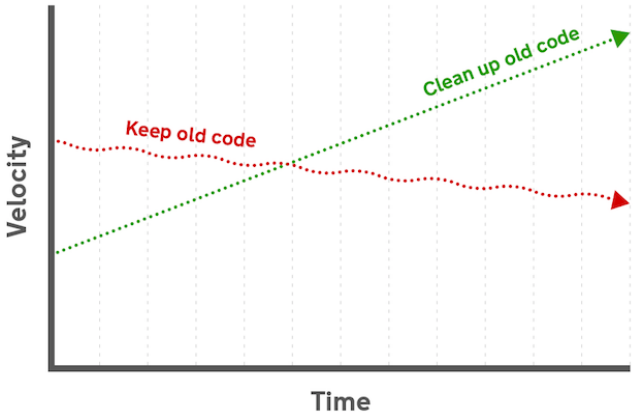


Fig. 5. Code quality v/s time[11]

Looking at all the approaches, we can clearly conclude that technical debt increases with time. The more we ignore the debt, the more it will accumulate and will be difficult to pay back later. Though the types of technical debt and the situation they are incurred in might be different but they always have negative effects on debt payback and maintainability. The ideal scenario should consist of cleaning up the code as development progresses, as illustrated in the figure above.

A way to reduce the occurrences of debt can be to introduce an automated tool that works on the basis of a set of rules. This tool can point out if a developer is going into debt based on real time and immediate inspection of the code. Since we have seen that the most frequent types of debt are also the easiest to fix, a tool like this can greatly help developers polish their code as much as they can during the programming phase. Though this tool will not be able to detect all types of technical debt, it would reduce the instances of debt by a lot. An example of the working of this tool can be: when a developer duplicates the same code, the tool flashes a warning sign on the screen to notify the developer. Another method can be to implement an auto-correcting software which will analyze the code and remove issues of debt without the developer needing to step in. Though this idea is not fool proof, it can alleviate the instances of simple and frequently occurring debts. Both of these ideas are abstract and needs further research, which we plan to carry out down the road. For now, it is evident that the issue of technical debt is here to stay unless developers become more aware of the flaws in their code and stick to the quality guidelines (or practice them on their own if not available).

ACKNOWLEDGEMENTS

We would like to thank Prof. R. Smedinga, Prof. M. Biehl and Prof. F. Kramer for giving us the required guidance in writing this paper.

REFERENCES

[1] Neil Ernst. A Field Study of Technical Debt. *SEI Blog, July 27, 2015 – Carnegie Mellon University.*

[2] Ward Cunningham. *OOPSLA Conference, 1992.*

[3] Tushar Sharma. Four Strategies for Managing Technical Debt. Retrieved from *http://www.designsmells.com/articles/four-strategies-for-managing-technical-debt/.*

[4] Agile Michael. Types of Technical Debt. Retrieved from *https://agilemichaeldougherty.wordpress.com/2015/07/24/types-of-technical-debt/.*

[5] Erik Frederick. The Financial Implications of Technical Debt. Retrieved from *https://www.toptal.com/finance/part-time-cfos/technical-debt.*

[6] Reducing Technical Debt. Retrieved from *https://xbosoft.com/software-quality-blog/reducing-technical-debt/.*

[7] Palomba, F., Bavota, G., Penta, M. et al. *Empir Software Eng* (2018) 23: 1188. https://doi.org/10.1007/s10664-017-9535-z.

[8] G. Bavota, B. Russo. A large-scale empirical study on self-admitted technical debt. *Proc. 13th Int. Workshop Mining Softw. Repositories*, pp. 315-326, 2016.

[9] Digkas G., Lungu M., Chatzigeorgiou A., Avgeriou P. (2017) The Evolution of Technical Debt in the Apache Ecosystem. *In: Lopes A., de Lemos R. (eds) Software Architecture. ECSA 2017. Lecture Notes in Computer Science*, vol 10475. Springer, Cham.

[10] Georgios Digkas, Mircea Lungu, Paris Avgeriou, Alexander Chatzigeorgiou, Apostolos Ampatzoglou. How Do Developers Fix Issues and Pay Back Technical Debt in the Apache Ecosystem? *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER).*

[11] Tekin Suleyman. Paying down (technical) debt in the departments and policy publishing platform. Retrieved from *https://insidegovuk.blog.gov.uk/2013/12/10/paying-down-technical-debt-in-the-departments-and-policy-publishing-platform/.*

An Analysis of Domain Specific Languages and Language-Oriented Programming

Lars Doorenbos and Abhisar Kaushal

Abstract— Conventional programming paradigms are not always congenial for developers to come up with efficient solutions, primarily because the existing languages provide a generalized approach towards solving problems rather than a specific approach to a particular field. Each domain has its own peculiarities and characteristics and a mainstream programming language might not be the best choice for developers working in that domain.
To overcome this, Language-Oriented Programming (LOP) comes into play, which involves first creating one or more Domain Specific Languages (DSLs) with which the problem will be tackled. Using such a DSL allows developers to focus better on domain specific tasks. Currently, DSLs embedded in a mainstream programming language are the norm. Although this modus operandi is an enrichment over traditional programming it comes with its own set of limitations, especially because it is embedded in a mainstream language.
In this paper we evaluate the advantages and disadvantages of using DSLs, using examples from the video processing and audio synthesis domains as a basis. We also shed light on the "middle-out" approach employed in LOP and how it differs from the traditional approaches. The main drawbacks include the cost of designing, implementing, and maintaining a DSL. These have to be weighed against the benefits, such as being able to represent a problem in the terms of the domain and the reduction in system complexity. We believe that if the domain and programming expertise are available, or can be obtained within reasonable time, LOP outperforms the other paradigms when it comes to the development of software systems.

Index Terms—Domain Specific Languages, Language-Oriented Programming, Audio Synthesis, Video Processing.

1 Introduction

Programmers are often required to code in a conventional programming language that was picked for them. As these languages are not always the optimal choice, recently software developers have started to increasingly develop and rely upon Domain Specific Languages (DSLs) [5]. These are languages created for limited application in a particular domain only. The motivation behind this trend is simple, working with a general purpose language to develop a specific product can be an inelegant and time consuming process, where the resulting product often can only be understood by its creator. Problems simply are more easily tackled with tools specifically designed for them. One of such tools is creating a DSL for the problem at hand. An example of this is jQuery which was developed to shorten lengthy JavaScript codes as well as handle events better and simplify creation of animations while working on web development. Conventional programming simply isn't always the most efficient choice for the job. This is well illustrated by database management systems, where using an object oriented language such as Java can make conceptually simple tasks complex. Instead a DSL named SQL is often used, which outclasses its counterparts that are implemented in a general purpose language. Currently, there are a lot of popular DSLs like SQL, MATLAB, LaTeX, etc. which provide functionality for certain essential and established fields like database management, numerical computing, word processing and the like.
It is clear that DSLs can provide an advantage over the conventional programming approaches given the right circumstances. It is however crucial to have a robust and refined paradigm to produce them. The Language Oriented Programming (LOP) paradigm has been the most prominent in tackling this challenge. Instead of programming the DSLs in a mainstream language, a language created with the LOP paradigm in mind is used as the basis for the DSLs. This approach to software creation and its benefits and drawbacks are the focus of this review paper.

- *Lars Doorenbos is a MSc Computing Science student at the University of Groningen, E-mail: l.j.doorenbos@student.rug.nl.*
- *Abhisar Kaushal is a MSc Computing Science student at the University of Groningen, E-mail: a.kaushal@student.rug.nl*

We start by describing the concept of LOP further in section 2, followed by a more detailed description of DSLs in section 3. We then take a look at two applications of DSLs in real world examples, namely the video processing and audio synthesis domains in section 4. In the end we try to generalize our findings and give some concrete guidelines as to when DSLs are applicable and when the more traditional approaches are a better fit.

2 Language-Oriented Programming

Language-Oriented Programming (LOP), introduced by Ward in 1994 [13], sets out to tackle the 4 problematic properties of large software systems described by Brooks [3], namely:

- The large complexity of the system,
- The need for it to conform to many institutions and systems,
- Its changeability in scope and time,
- Its invisibility or the difficulty of visualizing the system due to the lack of a geometrical representation.

As a result, LOP takes an unconventional approach to the development of a software system. Below we discuss the traditional approaches and how LOP differs from those. In Figure 1 an overview of the approaches discussed below is given.

The first method we discuss is the commonly used 'top-down' approach, where first a high-level description of the desired system is created, followed by slowly implementing all abstractions until the full product is reached. The problem with this method is that it is unknown beforehand how this top level structure should be designed. Especially for large systems this will create problems when the wrong structure is chosen, and these problems will only become apparent later on in the development process.
The opposite approach, 'bottom-up', begins with implementing the lowest level routines needed for the system and works its way up from there. The difficulty with this approach lies in the choice of which routines to implement next or how to figure out how lower-level routines fit into the larger system, where again wrong choices can significantly hamper the development process.

A combination of the two methods above is the 'outside-in' approach, where two teams start on the same system, one at the bottom on the lowest level routines, and the other at the top with the high-level description of the system. The goal is to meet somewhere in the middle where the fusion of both teams creates the final product. This method however incorporates the problems of both approaches, along with the added difficulty of 'meeting in the middle'.
LOP instead opts for the 'middle-out' approach. In this approach, first a domain-oriented, very high-level language (the DSL) is formally specified after which the development is split into two parts. The system has to be implemented using this new language, and a compiler, translator or interpreter has to be created for said language [13].

The 4 problems (complexity, conformity, changeability and invisibility) described above are addressed using this approach:

- The development process consists of two parts: the creation of a compiler, translator or interpreter for the DSL on the one hand and writing the program in this DSL on the other. Because of this the complexity of the system is reduced as the source code is split into two independent sections. To reduce complexity even further, multiple DSLs can be stacked recursively.

- Conformity is more easily achieved as the DSL uses the very concepts, such as the jargon, present in the institutions or systems the code should conform to.

- As the source code is smaller in size, changes are more easily made. Also due to the fact that the code is written in terms of the domain, these changes can in some situations even by made by the user.

- Invisibility is still a hard problem, but as some complexity is hidden in the language constructs, it will still be easier than in the traditional cases.

An example of a language specifically created with the LOP paradigm in mind is Racket. Its very goal was to create a programming language which enables language-oriented software design, such as the creation of DSLs [5].
The guiding principles of Racket state that they want to empower programmers to create new programming languages easily and to add them with a friction-free process to a code base. These are expressed by not only allowing the developers to create new languages independently and efficiently, but also by encapsulating attributes like a unique and interactive syntax, and a semantic system that maps the new syntax to elements in the host language and even foreign languages [5].

3 Domain Specific Languages

A Domain Specific Language (DSL), as the name specifies, is a programming language specifically designed for application in a particular field. As a clarification, this section will not deal with DSLs embedded in a mainstream language. These differ from a DSL developed with the LOP paradigm in mind as the former use the syntax of their mainstream host language.
In order to help understand DSLs better, let us first look at their life cycle:

- The first step, is that of domain analysis. Domain analysis originates from software reuse research, and can be used when constructing domain-specific reusable libraries, frameworks, languages, or product lines [11]. The goal of domain analysis is to come up with a feature model, which can contain a number of dependencies, similarities and variabilities between software family members and other variables. It may also contain business information like stakeholders or priorities.

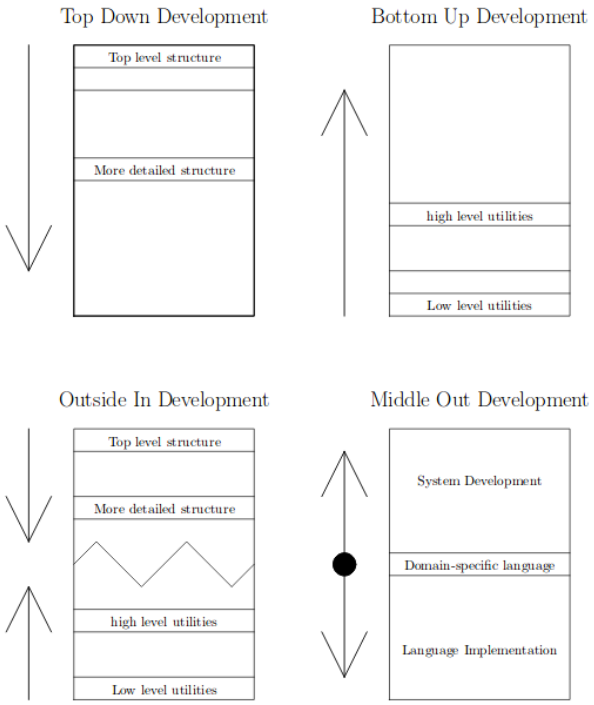


Fig. 1. Illustration of the different approaches to development [13].

- Once the domain has been analyzed, the software is designed according to the analysis. The design may or may not keep all the binding regulations defined in the analysis in mind. In some cases, the entire DSL designing may occur irrespective of the rules and regulations as they might hinder the imagination and capabilities of the developers. DSLs designed in this manner then need to undergo a strict evaluation session where much of their functionality might be eliminated.

- After the DSL design stands up to all the criteria in the domain analysis, the coding part can commence. This mostly concerns constructing a library that implements the semantic notions and designing and implementing a compiler that translates DSL programs to a sequence of library calls [12].

- Once the DSL prototype concisely describes the applications in the domain, it undergoes a rigorous testing session. The DSL can be tested on a number of parameters; usability, ease of understanding and execution, resource management, etc. If the DSL somehow fails in one or more tests, it needs to be re-evaluated and sent back to the designing step.

- Only when the DSL passes all the tests, is it qualified to be deployed. But, deployment is not a single faceted process. The DSL needs to re-evaluated periodically, frequently updated and thoroughly maintained.

3.1 DSL Implementation

A DSL can be implemented through an interpreter and/or compiler. Apart from using standard compiler tools, tools targeted strictly towards DSL implementation like Kephera, Draco, Kodiyak and InfoWiz can be used. Using these DSL specific compilers has an obvious advantage as they are better optimized to work on DSLs and no compromises are needed when it comes to semantics and notations. On the flip side designing these tools in the first place takes time not spent on designing the software system itself. Many, if not most

DSLs are limited to a particular application field and thus, the scope of the compiler/interpreter would be limited (although this is not the case with many tools which are designed while keeping multiple DSL implementation in mind).
Another cost effective and comparatively less tiresome approach is using already existing mechanisms, such as function definitions and/or operators with user-defined syntax. These can then be employed to build a library for domain specific tasks. The syntactic mechanisms of the base language are used to express the idiom of the domain [12].

4 DSL APPLICATIONS

In this section we analyze the application of DSLs in two domains, namely audio synthesis and video processing.

4.1 Audio synthesis

For the audio synthesis domain we take a look at the Bithoven composer [8]. Bithoven is a full-stack music synthesis system, automatically generating 8-bit music. It makes use of two DSLs in series, one used for the programming of digital instruments and one used for denoting the measures of the song. The DSLs are both implemented using Racket.

4.1.1 Digital instrument programming

The first DSL facilitates the programming of the digital instruments used in the compositions automatically generated by Bithoven. These instruments are simulated by generating and modifying particular predefined waveforms, consisting of pulse waves, triangle waves and a noise channel. The noise channel is used for the drum kit whereas the other two are used for the melodies. We consider 2 explicit examples below.
To program an instrument using the pulse channel, at least 3 parameters need to be given. These are the duty cycle, the period and the volume. The duty cycle determines in which parts of one period the signal will be active and in which it will remain silent. The period is used to specify which tone to play and the volume influences the strength of the output signal. A simple example of an instrument programmed with this pulse channel is shown in Listing 1.

```
(define (i:pulse:basic duty)
  (i:pulse
   #:duty (spec:constant duty)
   #:period (spec:constant 0)
   #:volume (spec:constant 7)))
```

Listing 1. A note played by a simple pulse channel instrument [8].

More details can be added to the programming of the instruments, which will be used to create differently sounding instruments, even when they play the same notes. This distinct sound of each instrument is called the timbre of the instrument.
A slightly more involved example of this is the programming of the bass drum as done for Bithoven, for which the code is shown in Listing 2. This uses the noise channel, which again takes at least 3 arguments, namely the volume, the period and the mode. The noise channel is either active or silent based on a pseudo-random number generator, and the period determines how often this generates a new number. The mode determines if the normal method is used or one which produces a more 'metallic' sound.
Added onto the basic parameters is the Attack-Decay-Sustain-Release (ADSR) specification. This divides the synthesis of each note into 4 stages. Generally, during the attack stage the signal of the note increases, then decreases in the decay stage, remains constant during the sustain stage and decreases again in the release stage. The ADSR specification is the main tool used for the creation of instruments with different timbres. In the example (Listing 2) we see the ADSR specifications for the noise channel given as the last parameter.
This first parameter of this ADSR specification states which stage will incorporate any leftover frames if the specification involves less frames than the duration of the note to be played. The following 4 parameters each define one of the 4 stages. The first number states how long that stage will last in frames, followed by what happens to the signal during that stage. In the example we see that the decay stage lasts 2 frames where the signal is at a constant 7, and the sustain stage drops from 4 to 2 over the course of 4 frames. Note that the signal strength does not need to strictly follow the regimen set by the stage names, it is free to for example remain constant during the decay stage.

```
(define i:drum:bass
  (i:noise
    #:mode (spec:constant #f)
    #:period (spec:constant 9)
    #:volume
    (spec:adsr 'release
               1 (spec:constant 10)
               2 (spec:constant 7)
               4 (spec:linear 4 2)
               4 (spec:constant 0))))
```

Listing 2. The programmed bass drum instrument [8].

Even more differently sounding instruments can be created by using the triangle wave channel, and more effects such as tremolo and vibrato can be achieved by giving non-linear parameters as inputs. These will not be discussed in this paper, and we refer to the original paper for further information on the musical as well as programming theory behind them [8].

4.1.2 Music tracking

The music tracker concerns itself with combining the audio tracks of the different instruments into a single song. In the case of Bithoven this results in combining 4 synthesized instruments together with a drum beat consisting of 3 noise instruments. It compiles a DSL and generates a corresponding sequence of synth frames that can be played by the synthesizer.
The DSL itself, when used with 4 instruments which is the case in Bithoven, closely resembles sheet music (and can be used to generate sheet music using Lilypond [10]). An example is given in Figure 2. Every row in said image consists of a fraction followed by 4 tones. Tones are given a names based on their pitch. These names consist of two parts. The first part is a letter indicating their position on the music scale, which ranges from A to G. The second part denotes the octave the tone is attached to, where a higher number represents a higher tone which has the same pitch. Each of these 4 tones is associated with one of the 4 instruments. These instruments will play their assigned tones for the time interval specified by the fraction at the start of each row. All rows are then converted in order into synth frames and the resulting list can be read by the synthesizer as a short song.

```
(((1/4 ((C 0) (C 0) (C 0) (C 0)))
  (1/4 ((D 0) (D 0) (D 0) (D 0)))
  (1/4 ((E 0) (E 0) (E 0) (E 0)))
  (1/4 ((F 0) (F 0) (F 0) (F 0))))
 ((1/4 ((G 0) (G 0) (G 0) (G 0)))
  (1/4 ((A 0) (A 0) (A 0) (A 0)))
  (1/4 ((B 0) (B 0) (B 0) (B 0)))
  (1/4 ((C 1) (C 1) (C 1) (C 1))))
 ((1/4 ((B 0) (B 0) (B 0) (B 0)))
  (1/4 ((A 0) (A 0) (A 0) (A 0)))
  (1/4 ((G 0) (G 0) (G 0) (G 0)))
  (1/4 ((F 0) (F 0) (F 0) (F 0))))
 ((1/4 ((E 0) (E 0) (E 0) (E 0)))
  (1/4 ((D 0) (D 0) (D 0) (D 0)))
  (1/4 ((C 0) (C 0) (C 0) (C 0)))
  (1/4 ((D 0) (D 0) (D 0) (D 0)))))
```

Fig. 2. A track playing a scale in C [8].

Similarly, the DSL can be used to represent drum beats. After specifying the 3 instruments which will simulate the hi-hat, the bass drum and the snare drum, their patterns for the measures are given. An example can be seen in Figure 3. The image shows that the first instrument will be played 8 times each measure at regular intervals, while the second and third instrument play 4 times at regular intervals.

```
(define beat:straight-rock
  '((1/8 1/8 1/8 1/8 1/8 1/8 1/8 1/8)
    (1/4 1/4 1/4 1/4)
    (1/4 1/4 1/4 1/4)))
```

Fig. 3. A simple drum beat [8].

4.2 Video processing

The conventional video editing process is a tedious task. Editors need to deal with different sequences, add animations and effects, detect and delete unwanted clips, and combine different tracks in order to produce an interactive finished product. The process is the same for each video, resulting in a monotonous and cumbersome routine.
As it turns out, the task of video editing naturally splits into a declarative phase and an imperative rendering phase at the end [1]. The DSL we analyze used for video editing, aptly named 'Video', embraces this concept. Each Video program is a module that is concerned with combining video descriptions as well as definitions like audio, animations etc. The Video language is implemented in and follows the doctrine of Racket and each Video module is a Racket module that exports a playlist description of the complete video.
To help understand this, let us consider the code in Figure 4. This code

```
01 #lang video
02
03 (image "splash.png" #:length 100)
04
05 (fade-transition #:length 50)
06
07 (multitrack (blank #f)
08             (composite-transition 0 0 1/4 1/4)
09             slides
10             (composite-transition 1/4 0 3/4 1)
11             presentation
12             (composite-transition 0 1/4 1/4 3/4)
13             (image "logo.png" #:length (producer-length talk)))
14
15 ; where
16 (define slides
17   (clip "slides05.MTS" #:start 2900 #:end 80000))
18
19 (define presentation
20   (playlist (clip "vid01.mp4")
21             (clip "vid02.mp4")
22             #:start 3900 #:end 36850))
23
24 (fade-transition #:length 50)
25
26 (image "splash.png" #:length 100)
```

Fig. 4. Descriptive phase using Video [1].

describes a total of five sequences, with multiple transitions within these fragments. Being the descriptive/declarative part of the editing process, the user can encapsulate this visual abstraction along with the audio support, to create a library. This library would further facilitate editing videos of the same kind. For example, an editor can define a library, 'lecture-lib.vid' which has predefined transitions and animations, along with the desired partitioning of the frame. This library can be used over and over without defining the modules repetitively, which makes the editing task less frustrating.
Let us now look into some of the features of Video in more detail:

4.2.1 Producers

Producers are the most basic entities in the Video programming language. They are analogous to data types in conventional programming. Producers are used to denote video clips, audio clips, images, etc. One of the most common producers is named 'clip', which fragments a video file into a sequence of frames.

4.2.2 Playlists

Playlists are the means to put together producers. Within the Video DSL, one may get confused by playlists and multitracks, as both are employed to combine producers. The difference being, playlists organize producers sequentially whereas multitracks do so in parallel. A small example is given in Figure 5.

```
#lang video

(playlist (clip "talk00.MTS")
          (clip "talk01.MTS"))
```

Fig. 5. Example of playlist combining two clips [1].

4.2.3 Transitions

Transitions are used to portray a smooth and appealing jump from one clip to another in a playlist. Transitions, thereby, also reduce the frame sequence in a playlist as two frames are converted to one with transition as a medium, see Figure 6.

```
#lang video

(image "splash.png" #:length 100)
(fade-transition #:length 50)
talk
(fade-transition #:length 50)
(image "splash.jpg" #:length 100)

; where
(define talk ⟨···elided···⟩)
```

Fig. 6. Using fade transition [1].

4.2.4 Multitracks

As stated earlier in the section concerning playlists, multitracks combine producers in a parallel manner. By parallel, it is implied that the multiple producers are executed simultaneously rather than sequentially. For example, an image and a clip can be both project simultaneously using multitracks. This is illustrated in Figure 7 where a clip and an audio track are combined.

```
#lang video

(multitrack
 (clip "slides.mp4")
 (composite-transition 0 0 1/4 1/4)
 talk)

; where
(define talk ⟨···elided···⟩)
```

Fig. 7. Multitrack combining a clip and audio [1].

4.2.5 Filters

Filters are similar to transitions, but they modify the behavior of only one producer. In other words, filters are functions from producers to producers. Among other effects, filters can remove the color from a clip or change a producers aspect ratio. Conference recordings frequently capture audio and video on separate tracks. Before splicing the tracks together, a developer may add an envelope filter to provide a fade effect for audio [1].

5 Discussion

We will discuss the benefits and drawbacks of using LOP and compare its effectiveness with the more traditional approaches.

5.1 Benefits

- A major benefit of using a DSL is the expression of problems in the language of the domain itself. This makes the code easier to understand, and ideally it can even be read and used by domain experts themselves without relying on the help of the developer of the software system. This is especially well demonstrated by the sheet-music like DSL discussed in subsubsection 4.1.2, which with minimal instructions should be readable and maybe usable for musicians.

- When properly created, this new language will also reduce the size and complexity of the system implementation. A very high level DSL will be able to express complex problems in a few lines of code. The Video language illustrates this well, where a few lines of code suffice for a problem such as creating a fade transition.

- With a smaller code base, maintenance will also inherently be easier [11].

- The designing process of a software system when using LOP is split into 2 parts, the design of the language and the system implementation using said language. A result of this is that the code of a software system implemented in this language is easily portable. Only the middle language in which the system is developed has to be ported, after which the implementation of the system, which is written in the ported language, can be copied without having to change it.

- When further problems in the same domain arise, well-designed DSLs can easily be reused. If a new synthesizer with a different amount of channels and properties would require programming, re-using the language defined in subsubsection 4.1.1 provides a better starting point than re-using a collection of previously defined data types and functions.

5.2 Drawbacks

- The flip side of the biggest advantage is the greatest disadvantage of LOP, the design of a good DSL can be very challenging. The programmer will need a good grasp of the problem domain and a solid understanding of the system requirements in order to be able to create a language for it, more so than in the traditional cases [11]. The examples discussed above were relatively simple and as such provide elegant solutions for the problems, but for more involved domains this is not as trivial. A solution for this is the recursive application of DSLs, where a previously developed DSL is involved in the creation of a new one. This can divide the problem into more manageable chunks when properly applied.

- Time has to be spent learning the new language instead of using one the programmer/client is familiar with. This drawback will be lessened the better the design of the DSL is, but there will always be an upfront learning cost. Tied in with this is the fact that the skills learned with respect to the DSL are generally applicable only in the domain for which it was written.

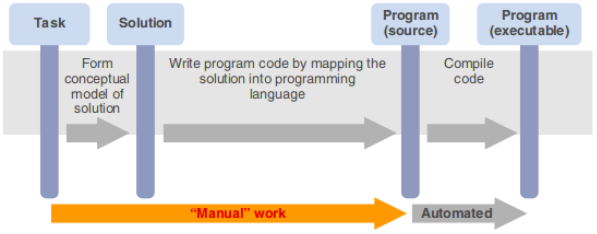


Fig. 8. Workflow when programming using a mainstream programming language [4]

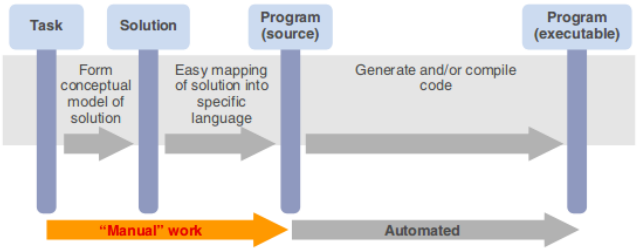


Fig. 9. Workflow when programming using a DSL [4]

- While the idea of having the domain experts being able to program the rules after minor training sounds appealing, for larger-scale projects the lack of technical expertise seems to prevent this. It was found to be more cost-effective still to train their own business analysts with technical background than to teach the end users how to use the new system. The end users were however mostly able to read the verification rules, and even sometimes modify them [6].

- Tool support such as debuggers for custom built languages have to be built from the ground up with said custom language, which might impede the speed of development and might frustrate programmers used to fully fledged IDEs [13][6]. There is however a variety of packages available made that can help with the creation of compilers, interpreters and language parsers.

5.3 Comparison with other paradigms

There is a conceptual difference between the programming process when using the LOP paradigm versus other paradigms [4]. The mainstream language workflow is illustrated in Figure 8 and the LOP one in Figure 9. These figures show that the mapping of the solution to a general programming language not designed specifically for that domain is the most time consuming step in the process. This exact step is reduced significantly with LOP due the the resemblance between the DSL and the problem domain language.
When looking at Table 1, we can see that DSLs have a higher productivity level than that of mainstream languages such as Java or C. This productivity level is taken from the SPR Programming Languages Table [7]. A higher productivity indicates that less statements are needed to achieve the same functionality. In general, if the productivity, measured in function points (FP), is doubled, the amount of statements needed is halved. Table 2 shows the relation between the level of the language and the average productivity of the staff working with said language.
The 'middle-out' approach taken by LOP solves the issues plaguing the other approaches often employed by the different paradigms. Examples include the need to have an overview of the whole structure of the system to be designed when using the 'top-down' approach, often used for the imperative programming paradigm with languages such as C, as well as figuring out how lower level functions fit into the bigger picture, often seen when using Object-Oriented Programming with languages such as Java. See section 2 for a more detailed

Language	Application domain	Productivity level
Excel	Spreadsheets	57
HTML	Hypertext web pages	22
Make	Software building	15
SQL	Database queries	25
VHDL	Hardware design	17
Java	General purpose	6
C	General purpose	2.5
Haskell	General purpose	8.5

Table 1. Table contrasting productivity levels between popular DSLs and mainstream languages [9] [7].

Productivity Level	Average productivity per Staff month (FP)
1-3	5-10
4-8	10-20
9-15	16-23
16-23	15-30
24-55	30-50
>55	40-100

Table 2. Table depicting relation between language level and average productivity [9].

description of the approaches and their problems.
Most students and developers working these days will have very limited experience with LOP. LOP is barely mentioned in papers discussing which paradigms to teach to students, see for example [2], and it is often mentioned how the advantages of LOP are unnoticed by current developers, see e.g. [9]. This implies that the chance the developers considering using LOP have an understanding of these concepts is slim. As a result, developers either have to be taught these new concepts, or outside forces have to be hired for the development process. Either way, this can be a costly and time consuming process not present when using a mainstream programming paradigm.
Equivalently, the developers do not require as much knowledge of the domain when using mainstream programming paradigms compared to LOP. This again would require an upfront cost of teaching the programmers about the domain if the LOP approach is chosen.

With all of the above in mind we can give some concrete notes on when we believe using the LOP paradigm over other paradigms is beneficial when designing a software system.

- The required domain and programming knowledge are available, or if the extra training in these fields required for the developer(s) is small; if the upfront training needed pays off in the long run.
- The DSL to be developed can be used in multiple places in the software system or in other places.
- The system is expected to be complex and frequently changing, in which case the benefits of DSLs such as a lower code complexity and easier maintenance shine [6].
- The domain experts have to interact with the code base, as the expression of the problems in the domain jargon will reduce the training cost necessary.

Of course all of the above points are trade-offs, there is some break-even point where for example training domain experts or training the developers becomes worth it in the long run. Determining this in advance, where the decision to use LOP or not should be made, is very difficult.

6 Conclusion

The benefits of the LOP paradigm, such as improved development productivity, flexibility, maintainability, and separation of business and technical aspects can outshine the conventional approaches and the use of general purpose programming languages. Nonetheless, there are also drawbacks such as the task of designing compilers and other tools necessary, and there is evidence that DSLs are not always successful when it comes to developing large-scale information systems. LOP is not the end-all solution to every problem. We do however believe that if the required expertise for the design of a complex software system, regarding both the domain and the programming of new languages, is present or can be obtained within reasonable time, LOP will outperform the more traditional programming paradigms.

References

[1] L. Andersen, S. Chang, and M. Felleisen. Super 8 languages for making movies (functional pearl). *Proceedings of the ACM on Programming Languages*, 1(ICFP):30, 2017.

[2] E. Bolshakova. Programming paradigms in computer science education. 2005.

[3] F. P. Brooks. No silver bullet essence and accidents of software engineering. *Computer*, 20:10–19, 1986.

[4] S. Dmitriev. Language oriented programming: The next programming paradigm. *JetBrains onBoard*, 1(2):1–13, 2004.

[5] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, 2018.

[6] M. Freudenthal. Domain-specific languages in a customs information system. *IEEE software*, (2):65–71, 2010.

[7] C. Jones. Spr programming languages table release 8.2, 1996.

[8] J. McCarthy. Bithoven: Gödel encoding of chamber music and functional 8-bit audio synthesis. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*, pages 1–7. ACM, 2016.

[9] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.

[10] H.-W. Nienhuys and J. Nieuwenhuizen. Lilypond, a system for automated music engraving. In *Proceedings of the XIV Colloquium on Musical Informatics (XIV CIM 2003)*, volume 1, pages 167–171, 2003.

[11] A. Van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[12] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, 2000.

[13] M. P. Ward. Language-oriented programming. *Software-Concepts and Tools*, 15(4):147–161, 1994.

A Brief History of Concurrency: Lessons from Three Turing Lectures

Michael Yuja and Bogdan Bernovici

Abstract—This paper covers the lectures given by three Turing award winners, Edsger Dijkstra, Robin Milner, and Leslie Lamport. They discuss personal experiences as well as their seminal work in the fields of concurrency and software correctness. Dijkstra accounts his experience of the growth of programming as a profession, and highlights his insistence on approaching programming with great humility and appreciation of its difficulty. Milner presents his work on the Calculus of Communicating Systems (CCS) used to formally model concurrent algorithms. Lamport's work is an extension of these ideas into more practical applications of concurrent algorithms. He built PlusCal and TLA+, which are programming languages used to write algorithms. Although separated by many years, they built on each other's ideas to shape the modern field of concurrency. Their work has paved the way for the industry to start adopting concurrent algorithm verification in practice, but there is not yet a widespread adoption.

Index Terms—concurrency, Dijkstra, Lamport, Milner, software correctness, fault-tolerance

1 Introduction

The field of concurrency in computer science has grown to play an important part in modern applications. Its growth can be markedly attributed to a select few computer scientists since its inception in the 1960's. In this paper, we will review the works of three exceptional computer scientists. They are recipients of the Association for Computing Machinery's (ACM) A.M. Turing award, which is often regarded as the Nobel Prize in computer science. We examine the lessons offered by Edsger Dijkstra, Robin Milner, and Leslie Lamport upon reception of the award, each of whom won the award in 1972, 1991 and 2013, respectively. The vast range in years between these publications allows us to give the reader a historical overview of how each scientist influenced the other. We give an evolutionary view of concurrency and its applications in practice today.

The definition of concurrency itself has changed over time. As Lamport suggests, concurrency in literature has been called many names: parallel computing, concurrent programming, multiprogramming, and distributed computing. For the most part, these all refer to the same thing, with some subtle differences. In the 1960's, as Lamport writes, research about concurrency was only about creating language constructs for concurrent programs. For this reason, he attributed Dijkstra to have created the field of concurrency as we know it today through his seminal paper that introduced the concept of mutual exclusion. If you think about how computer programs work, you will quickly come to the conclusion that computers follow a series of sequential instructions. Two different processes could indeed be executed at the same time, but this may become a problem when the computer's resources, such as memory, are being shared. One process could attempt to read from the same memory location that another process has just modified, although this behavior was not intended. The concept of mutual exclusion posed the main problem of concurrency by describing such a situation in which N processes must synchronize, such that no critical parts of the processes are executed concurrently and that all processes do get executed eventually. The purpose of mutual exclusion, Lamport states, is actually to eliminate concurrency. Programmers must eliminate, by design, the possibility for processes to be unable to synchronize.

Concurrency is a complex concept to apply in programming. Dijkstra, Milner and Lamport all recognized this. Dijkstra, for example, stressed that programming by itself was already hard enough, and even much more once programs were being written to run concurrently. He understood that it was almost impossible for a programmer to try to think like a computer because both are different from each other, and doing so would usually result in buggy software. Instead, he argues, a programmer should attempt to write programs correctly without introducing any bugs in the first place. Influenced by this line of thinking, Milner saw the need for a formal specification language that allowed algorithms to be written and checked for correctness. This resulted in his creation of the Calculus for Communicating Systems, or CCS, a language used to model the behavior of concurrent systems. We go over the principles of CCS in detail in Section 4. CCS established a formalism for concurrency. Lamport then carried on with the work of giving the semantics of concurrency more practical applications. He devised several improvements on concurrent algorithms with solutions to Dijkstra's problem of mutual exclusion, and conceived a new language for writing algorithms called PlusCal.

In this paper, we first explain the methodology we used to choose these papers and others that are worth mentioning in this study in Section 2. After, we dive into the contributions from each scientist individually. We review Dijkstra's lecture in Section 3, Milner's paper in Section 4, and Lamport's article in Section 5. Once we explain each author's individual contribution, we examine the underlying arc that connects the three in Section 6. Finally, we conclude our review in Section 7.

2 Methodology

These papers represent the culmination of decades of work by each author. As such, it would not do the authors justice for us to merely compare their ideas without an in-depth review of their own. Furthermore, we have chosen to review each paper individually, extracting the most important lessons from each one. We then compare their ideas with the present, exploring which are pervasive throughout the field, and which have not yet been adopted by the scientific community. It is indeed difficult to find where these practices are being applied, as such practices are usually kept secret within companies so that their trade secrets are not given away. It is only when a company decides to publish some of their work as open source that we can get an insight as to what they are doing, but usually these publications are a couple of years behind on what the company is actually doing at the time.

3 Dijkstra: The Humble Programmer (1972)

Upon reception of his award, Edsger Dijkstra offers a lecture to the audience in which he details a unique account of the evolution of the programming profession from his perspective [1]. In the introduction to Dijkstra's talk, M.D. McIlroy acknowledges that Dijkstra is recognized by the ACM for his approach to programming as a high, intellec-

- *Michael Yuja is a Master's student at the University of Groningen, E-mail: m.j.yuja.matute@student.rug.nl.*
- *Bogdan Bernovici is a Master's student at the University of Groningen, E-mail: b.g.bernovici@student.rug.nl.*

tual challenge, insisting that programs should be composed correctly, and not just debugged into correctness. Dijkstra reaffirms this idea in his talk with multiple examples that, albeit silly, offer a refreshing historical insight into how the performance improvement in computer hardware created entirely new opportunities for programmers. Finally, Dijkstra concludes his lecture with the idea that a programmer should approach the job with humility, hinting that doing so will keep the programmer from writing poor quality software.

3.1 A New Field

Edsger Dijkstra recounts the story of his career as a generalization of how programming slowly grew in other parts of the world as well. In the 1950s, he started carried out his studies as a theoretical physicist. After a chat with his boss at the Mathematical Centre in Amsterdam, where he had been programming for some years, he swiftly changed careers. He became the first programmer in the Netherlands. The profession was so unknown at the time that when he married in 1957 the Dutch authorities did not allow him write his profession on the marriage certificate as "programmer," arguing that it did not exist. As such, he was forced to list his profession as being "theoretical physicist."

Computer hardware had all the attention in the early years, while programmers were barely noticed. Dijkstra argues that people were much more impressed by these machines that would fill up entire rooms than by sheets of code. He believes that this view would later form two distinct opinions of programmers that would affect the field for many years to come. One opinion is that a competent programmer should be puzzle-minded and fond of clever tricks; the other is that programming was only about optimizing the efficiency of computations. According to Dijkstra, such opinions made people believe that programming would disappear once computers got faster. Years later, Dijkstra was proven right. Once computers did get faster, the need for programming grew even more. Dijkstra states that the increased power of hardware allowed programmers to think about solutions that were not feasible to even dream about only years before.

3.2 The Humble Programmer

Throughout the lecture, Dijkstra paints a picture of what a competent programmer should behave as. He demands that programmers must not waste time in debugging programs, but they should not introduce the bugs to begin with. This idea is recurrent throughout the examples and reasons that he gives for doing so. Dijkstra goes on to call for a revolution of sorts in which all programmers in the field strive to take on this intellectual challenge. As a funny, yet worrisome real world example of how programmers do not do this, he cites the role of "one-liners" in the workplace. Programmers write one line computer programs and proudly ask their colleagues to guess what they do, or to write an even shorter one that does the same task. This practice will likely lead to write programs that are so concise that they are too complex to comprehend at first glance. Programming is already a difficult task, Dijkstra argues, so there is no need to make it harder.

4 MILNER: ELEMENTS OF INTERACTION (1991)

Winner of the Turing Award in 1991, Robin Milner [4], reflects on a possible path of research in semantic basis of concurrent computation from a personal point of view. He begins his lecture with a fresh approach on concurrency, and culminates by describing his endeavour in finding basic constructions for concurrency. The latter being the efforts from which Calculus for Communicating Systems (CCS) was born.

In the beginning of his paper, he clearly rejects the idea of a unique conceptual model because of the vast pool of applications that have concurrent computation at their core. He calls for the necessity of having many levels of explanation which consists of a variety of languages and formalism to cover a broad spectrum of specialisms. But he also recognizes the nature of the computer scientist, being the one that is always looking for a unified framework from which you can expand all of those levels of explanation described above.

Although when it comes to the micro cosmos of sequential computation, we already have a common semantic framework that is generally based on the notion of a mathematical function and it is formally expressed in a functional calculus, he recognizes that for the macro cosmos of concurrent programming and interactive systems, the situation is different, there is nothing similar to the former.

Milner's work is split in seven parts, each of which is a semantic ingredient of concurrency. These will be discussed in summary below.

4.1 Entities

Milner begins by briefly describing the process of representing a sequential program, which is a function that maps memories to memories. He continues with a discussion of how λ-calculus's domain gives the meaning of an imperative program, which is based on the theory of domains, developed by Dana Scott. In addition, he argues about the compositionality enabled by the domain in sequential languages, such that component programs can give meaning to a composite program. Concurrency comes with non-determinism and compositionality is lost when combined subprograms are running in parallel. In sequential programming non-determinism is solved by using power domains, but when it comes to concurrency, the compositionality loss is still a reality. As a solution, Milner thinks about developing a general model of interactive systems, because the memory is no longer at the hand of a single master. Before, in the sequential world, memory was seen as monolithic but in the concurrent one, various parts of memory is accessed simultaneously. He regards each cell of memory as a process, linked to one or many programs which are also, themselves, processes.

Finally, he recognizes that shared memory is helping software engineers to accomplish their tasks by offering a level of correctness in their programs but for some use cases, it does not accommodate. Shared-memory model is an essential model for an engineer, but for a computer scientist, a unified theory of the ingredients is more sound. All interactions must be treated the same, hence the name of the paper, "Elements of Interactions".

4.2 Static Constructions

When it comes to primitive construction, Milner tells us that we need a fresh approach. He advocates that we should not add extra material to the languages and theories of sequential computing. By limiting ourselves to a set of constructions that are essential for concurrency, only then we can see sequential computing as a higher level of explanation. For him, functional calculus was just a paradigm and not a platform for building a calculus for communicating systems.

Milner came to that conclusion when he discovered that sequential composition of processes is a special case of parallel composition. In CCS, there is only a single combinator for combining processes. Even memory registers are modeled as processes in such a way that the same combinator can be used to put them into a memory, to compose the processes which use them, and to combine processes with memory. With a single combinator, the algebraic nature of calculus emerges, and we can envision components of a system being assembled together.

4.3 Dynamic Constructions

The author begins this subsection by contrasting the sequential control of the λ-calculus with the concurrent control of CCS. In λ-calculus, we have reduction, while in CCS we have interactions. The former is the action of passing an argument to a function, and the latter is the passage of a single datum between processes.

Furthermore, in each calculus, systems are constructed using a binary combinator. For λ-calculus we call it application, the dynamics of function application, which it is neither commutative nor associative. In the case of CCS, we have channels. The symbol λ, which represented the single unit of control is now one of the many channels. Parallel composition is commutative where either partner can act as a receiver or a transmitter. Moreover, it is also associative.The core idea of this ingredient is the synchronized interaction as a programming primitive, expressed in an algebraic form.

4.4 Meaning

Milner discusses about the dynamics of interaction between processes. He supports the idea that observing a process is simply put, interacting with it. If we cannot distinguish them by observation, only then we can use the same denotation for these process terms. Moreover, the remarks discussed earlier also apply to sequential computing. As you may already expect, however, concurrency comes with its own problems. The concept of *causal independence* where two processes may look indistinguishable and yet differ in their causal independence, can be problematic. Milner also brings into discussion Nielsen's *event structures* but does not dive into it because of the topic's complexity. He closes the section by suggesting that process calculi gives an essential perspective for the study.

4.5 Values

Milner compares λ-calculus to CCS. He argues that λ-calculus achieves homogeneity and completeness. Because λ-calculus does not provide a self-contained model of computing, everything can be done with scaffolding. Moving on to CCS, Milner suggested that even if it has a form of scaffolding, the promiscuity is excessive, it contains things like: processes, channels, variables, operators, value expressions. He believes that a basic calculus should impose as little taxonomy as it can. On the opposite side, λ-calculus has two things: terms and variables. He concludes by mentioning Carl Hewitt's Actor model which thought of a value, an operator on values and a process as being of the same kind of thing, an actor. The actor implies homogeneity and completeness.

4.6 Names

While talking about this ingredient of concurrency, Milner, puts emphasis on π-calculus, having naming or reference at its core, claiming several reasons for its generality. Data structures can be defined uniformly with π-calculus, supporting CCS as a higher level of explanation. Moreover, it supports functional programming as a higher level of explanation, demonstrated by a translation of λ-calculus into π-calculus. In addition, it provides semantics for object-oriented programming and other programming paradigms. π-calculus permits little taxonomies, similar to λ-calculus.

4.7 Milner's contribution

One particular characteristic of Milner is that he always strove for simplicity. The way he thought of processes, communication and synchronization ignited a wave of research into concurrency. CCS, born out of Process Calculus, paved the way for π-calculus and CSP, which is the more powerful equivalent of CCS. As a parentheses, the development of CSP into a process algebra was influenced by Milner's work on CCS (1980), and vice versa. This influence is, in fact, interesting because Tony Hoare, the creator of CSP (introduced in 1978), cites only Dijkstra in his work. Although they were similar in their objectives, there were no mainstream programming languages that were developed based on π-calculus or CCS, unlike the λ-calculus which went on to inspire very popular languages like LISP. Some ideas from CSP can still be found in today's programming languages like Erlang, Scala, Clojure, Go, etc.

5 Lamport: The Computer Science of Concurrency (2015)

Leslie Lamport won the Turing award in 2013 for his contributions in concurrent and distributed computing [2]. In his paper, he applies a methodology similar to what we are doing in our review. He gives a brief history of how the main ideas of concurrency evolved since Dijkstra's introduction of the mutual exclusion problem in 1965 up until 2015 when his paper was published after receiving his award. Lamport cites Dijkstra as being the only computer scientist in the late 1960's who was actually working on what we know today as concurrency. This paper will revisit and briefly discuss each step that Lamport considered as a worth mentioning achievement in the history of concurrency.

5.1 Mutual Exclusion

Mutual Exclusion is the goal of eliminating concurrency. It means that two critical sections must not be executed concurrently. It was formally defined in 1985 as a *safety* property. In the same manner, another property called liveness emerged in the same year. Both mutual exclusion (safety) and livelock freedom (liveness) were described by Edgar Dijkstra, in his paper from 1965. In this context, he proposes a problem with N processes that need to be synchronized. Every process having a critical section, such that the properties we have presented above are satisfied. Lamport highlights the computation model used by Dijkstra consists of a sequence of states, each state having attached to it an assignment of values to the algorithm's variables, together with other information specific to the state (like what should be executed next). Lamport prefers to call it the *standard model*. After the second algorithm has been proved to be incorrect, the need for careful proofs became apparent. The Bakery algorithm is the most popular example of a mutual exclusion algorithm. Imagine those support centers that have machines with consecutive numbered tickets for their customers. The ticket numbers are not subject to a finite limit, even if it doesn't seem like a practical problem, a ticket number can hold more than one memory word and it was assumed that the process can do read and write operations on at most one word.

The proof of correctness for Bakery algorithm demonstrated that it doesn't matter if a read overlaps a write. Reading a number when it is incremented from 9 to 10 can give a result like 2496, which still proves that the algorithm is correct. In 1973, it was considered impossible to assume that mutual exclusion can be implemented without any lower-level mutual exclusion. Lamport continues his discussion on the topic of mutual exclusion with a more rigorous proof. Putting aside the standard model, he introduces us to a model that assumes atomic transitions between states, one that does not provide a natural model of the bakery algorithm. Furthermore, before diving to the newly proposed model, he exposes us to a fundamental problem of inter-process synchronization which is guaranteeing that an operation executed by one process precedes an operation executed by another.

The *two-arrow model* is a set of operation executions with a considerably finite duration with both, starting and stopping times. For a proper explanation, Lamport described the operation executions A and B as follows:

- $A \rightarrow B$ is true iff A ends before B begins.
- $A \Rightarrow B$ is true iff A begins before B ends.

For any operation executions on A, B, C, D, the relations from above should satisfy the following properties:

1. $A \rightarrow B \rightarrow C$ implies $A \rightarrow C$ ($\rightarrow$ transiently closed)
2. $A \rightarrow B$ implies $A \Rightarrow B$ and B not $\Rightarrow A$
3. $A \rightarrow B \Rightarrow C$ or $A \Rightarrow B \rightarrow C$ implies $A \Rightarrow C$
4. $A \rightarrow B \Rightarrow C \rightarrow D$ implies $A \rightarrow D$

The following assumptions are taken into account to prove the correctness of the algorithm:

- operation executions are totally ordered by $\rightarrow$
- for a variable that is being read R or written W, either $R \Rightarrow W$ or $W \rightarrow R$ holds

According to Leslie Lamport, together with the assumptions specified above, the two-arrow formalism gives the most elegant proof of the bakery algorithm.

5.2 Producer-Consumer Synchronization (FIFO queue)

Producer-consumer synchronization is the second fundamental problem in concurrent programming. The algorithm is using three variables: *in* for the unbounded sequence of unread input values, *buf* as a buffer that can store up to N values and *out*, the output sequence of values. The *Producer* process takes values from *in* and puts them into *buf* and the Consumer process, moves them from *buf* to *out*. After showcasing the algorithm in PlusCal, he argues that it is just a specification, a mere definition and there is no sense in finding out if a definition is correct. To increase our confidence that the algorithm actually implements a FIFO queue, we have to prove properties of it. Invariant properties being the most important, thus should be proven. The author makes parallels between mutual exclusion and producer-consumer synchronization, the former being inherently nondeterministic and the latter, deterministic. In the case of mutual exclusion, there is an inherent race condition that needs to be addressed by using an arbiter. Lamport makes it evident that producer-consumer synchronization is a different class of problem than mutual exclusion.

5.3 Distributed Algorithms

Lamport starts by highlighting pictures of event histories. These have been used in the past to describe DS. Mainly, the events come from three processes, with time moving upwards. Lamport argues that even if event histories may be useful in understanding distributed systems, the best technique to reason about them is in terms of global invariants. There is no more practical way to reason about invariance than the standard model which he has previously described.

Lamport then explains that fault-tolerance was not a very popular topic in the 1970's. He cites Dijkstra's paper on self-stabilization as the first scientific examination of fault tolerance. Lamport praises Dijkstra once again by saying that Dijkstra and his ideas were ahead of this time, and that this is the reason they were not adopted during that period. For many years, fault tolerance was stagnated, and Lamport references Milner's CCS as the only influential work in the study of models of concurrency that emerged in the 1980's.

6 Concurrency Today

The study of concurrency has certainly grown in importance over time. The field has expanded to have a direct effect in both how hardware is manufactured and how software is written. Early on, Dijkstra predicted the need for thinking about concurrency in both hardware and software, contrary to popular belief at the time. The contributions by all the scientists in this field have led to the development of concurrent algorithms which are widely used in practice today. Concurrent algorithms allow modern devices such as smartphones to give the illusion to the end user that all its applications are being executed in parallel.

6.1 The Difficulty Persists

Throughout the years of growth in the field, however, one major hurdle persists: writing concurrent programs is still very hard. The reality is that many inefficiencies in programming, as Dijkstra points out, are usually made up by improvements in computer hardware. He argues that the efficiency of programming as a job should improve at the same pace as hardware does. In practice, however, the tools and methods used to verify the correctness of programs, such as those described in the previous sections, have not been widely adopted by modern technologies and engineers. As Lamport wrote in a recent article, "architects draw detailed plans before a brick is laid or a nail is hammered. But few programmers write even a rough sketch of what their programs will do before they start coding." Furthermore, with the exponential increase in demand of software developers, companies are willing to hire more self-taught developers that do not have the formal training and mathematical background that computer scientists do. This trend would suggest that we are actually moving further away from the future that Dijkstra envisioned in the 70's. [3]

6.2 Practical Applications of Formal Modelling

Although it may seem like the future is bleak for applying mathematical rigour to programming, we must acknowledge that scientists such as Lamport continue to carry the practice forward. Technology firms do recognize the business value of doing so in their software engineering practices for critical systems. Lamport's TLA+, a language similar to PlusCal, has been used in several industrial settings, such as by the Amazon Web Services team in the S3 service to prevent serious but subtle bugs from reaching production. [5] For now, the use of TLA+ as a verification language has found its way to niche industrial uses by large companies. The vision, however, is to steward the practice to acceptance by the entire development community.

TLA+ was primarily made to be used by ordinary engineers, not formal method experts or logicians. In practice, the software engineers have to make trade offs, and find a common ground between academia and industry. As Leslie Lamport said, "for quite a while, I've been disturbed by the emphasis on language in computer science... I believe that the best way to get better programs is to teach programmers how to think better. Thinking is not the ability to manipulate language; it's the ability to manipulate concepts." In addition, the tools that come with TLA+ place it as a strong contender for our needs. First of all, you get IDE support. Secondly, there is a model checker called TLC that takes the description of your algorithms and verifies your assumptions about it. Lastly, for formal proofs, there is a proof language with a interactive proof assistant called TLAPS.

7 Conclusion

We have reviewed ideas from three influential computer scientists and extracted the most important ideas and contributions to the field. Dijkstra was adamant in changing the way that programmers write software. He urged his audience to accept that programming is difficult, and that one part of the programmer's job is to find ways to write better software. Software engineers should strive to write programs correctly in the first place, rather than debug them into correctness. We have seen that this behavior is still not completely prevalent in industry, and that trends may suggest we are even moving away from it. Methods of formal verification such as Milner's CCS and Lamport's TLA+ carried on with the right intentions, but have failed to achieve mass adoption by software engineers.

It is the ideas that Dijkstra, Milner, and Lamport have contributed that are paving the way for widely accepted practical applications of concurrent program correctness. It is not clear whether PlusCal or CCS or any of the other variations will become prevalent in practical use, but it is certain that there must be a fundamental change to how developers write software. As Dijkstra wrote, "the purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise."[1] This idea was the underlying principle by which CCS was created, and it will continue to play an important role in whatever specification language the community adopts. We can envision tools embedded in integrated development environments or as part of languages that allow developers to apply mathematical rigour to their software. As with any disruptive technology, widespread adoption by users will become exponential once the technology is able to serve the majority of users. We invite the reader to tackle this problem with creativity and ingenuity, as doing so successfully would likely result in a great milestone to the field and an even greater business opportunity.

References

[1] E. W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, Oct. 1972.

[2] L. Lamport. Turing lecture: The computer science of concurrency: The early years. *Commun. ACM*, 58(6):71–76, May 2015.

[3] L. Lamport. Who builds a house without drawing blueprints? *Commun. ACM*, 58(4):38–41, Mar. 2015.

[4] R. Milner. Elements of interaction: Turing award lecture. *Commun. ACM*, 36(1):78–89, Jan. 1993.

[5] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff. How amazon web services uses formal methods. *Commun. ACM*, 58(4):66–73, Mar. 2015.

Selecting a Logic System for Compliance Regulations

Michaël P. van de Weerd and Zhang Yuanqing

Abstract— Changes in regulations present a challenge to businesses to maintain compliance within business processes. Compliance regulation verification (CRV) systems provide the means to ensure that business processes are up-to-date according to current regulations. These systems require regulations to be expressed using a compliance request language (CRL), of which several are available. In this paper, we present an addition to the process of expressing regulations using a CRL by providing a work flow for selecting the proper CRL for a set of regulations. To this end, we propose a framework that can be used to analyze regulations and identify their properties. Combining these properties and the requirements for CRLs the appropriate CRL can be selected. We demonstrate the use of the framework and work flow with a case study, using a set of fabricated Starfleet regulations.

Index Terms—business process, temporal logic, deontic logic, compliance verification

1 Introduction

In the modern day and age, businesses are expected to comply to a plethora of regulations — e.g. governmental laws, internal policies, etc. — referred to as *norms*. As a result, organizations are presented with the challenge to keep up with the ever changing set of rules that must be obeyed. To solve this problem, many have turned to the domain of the automation of CRV to find conflicts between business processes and norms during the design process, process execution or after the fact using execution logs. While the currently available literature provides us with a lot of theoretical knowledge on writing compliance constraints, business processes and utilizing logic systems (see [3], [4], [6], [7] and [8]), the field lacks a methodical approach to analyze the properties of compliance constraints and selecting a logical system in order to express them. In Figure 1 an illustration of the process of CRV has been included to indicate the position of the problem we have identified.

In this paper, we explore the field of formulating norms to give an overview of the options available to any modern business regarding logical systems in which the regulations can be formulated. Using a formal definition of a compliance regulation and its properties, we will provide guidelines that will allow businesses to match their set of regulations against the different logical systems. Ultimately, we hope to contribute to the field of writing compliance regulations by providing a framework that can be used to analyze a collection of compliance constraints in order to match it to a suitable logical system. This goal assumes that the user — e.g. a business developer — has access to a complete set of compliance regulations and has the ability to analyze their properties as described in section 5.

Before diving into the specifics of selecting a logic system for a set of compliance regulations, we will give an overview of related work that shape the current knowledge of the subject in section 2. Next, more background knowledge on logic systems is provided in section 3. A detailed description of our research method is included in section 4. In order to allow for a methodical approach to selecting a logical system that matches a given set of compliance regulations, we will present a formal definition of a compliance regulations and their properties in section 5. Next, in section 6, we will review the definition and properties of the logical systems as presented in other literature. Linking the properties of both concepts in section 8, we will combine the gathered knowledge and propose a set of logical steps to consider when selecting a logical system for specific use-cases. In section 9 we will review the applicability of our own findings and propose opportunities for further research.

Michaël P. van de Weerd and Zhang Yuanqing are with the University of Groningen. E-mail: {m.p.van.de.weerd, y.zhang.109}@student.rug.nl.

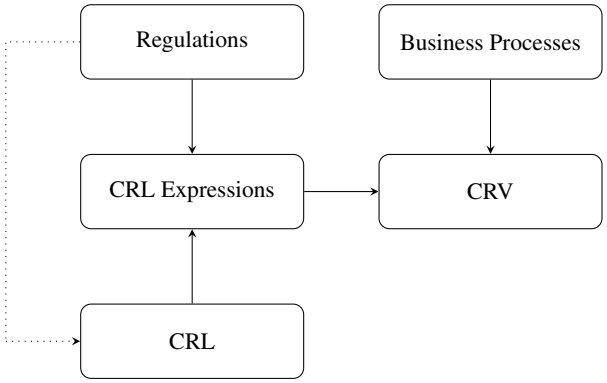


Fig. 1. A flow diagram illustrating the complete process of verifying compliance to regulations. The aim of this paper is providing a framework that can be used to analyze the textual regulations in order to select a proper CRL. To this end, the process needs to be extended with the process indicated with the dashed line.

2 Related Work

A fair amount of theory on logical systems is available. In this paper, we will consider three types of logical systems: Formal Contract Language (FCL) [5], Linear Temporal Logic (LTL) [10] and Computational Tree Logic (CTL) [3], all described as a basis for CRLs to be build upon. This selection has been made to reflect the selection of CTLs considered in [3]. In [3] we find a concise use-case analysis of these systems, in which each is compared to the other and their strengths and weaknesses are identified. Also, [3] describes the two families of logical systems: *deontic logic* and *temporal logic*, the prior of which encapsulates FCL and the latter LTL and CTL. [4] presents us with a technical description of the manner in which these system can be utilized. Both [8] and [3] provide us with a description of the properties of compliance regulations, although they are not listed explicitly. Most of these properties are directly or indirectly related to the requirements for CRLs as mentioned in [3]. In our research, we are only interested in the *non-functional properties* of compliance regulations. We define this concept as being analogous to the concept of *non-functional requirements* used in computer architecture, i.e. properties that indicate what the required *capability* of the system, without necessarily providing any information about *functionality* [2].

In this paper, we frequently employ the term *use case*. In the context of our research, a use case has been defined as a collection of compliance constraints. We will make use of the following definition of compliance constraint, presented in [3]:

> "A norm is a statement by a body/entity (with the authority or power to create, and eventually to enforce, such statements) prescribing or regulating some behaviours such that

> the non-adherence to the norms potentially leads to some sanctions."

N.b., the terms *compliance constraint*, *compliance regulation*, *norm*, *rule*, *law* and any variation thereof are used interchangeably. As mentioned in section 1, we assume that the *user* — e.g. a business analyst — has collected their compliance constraints by their own means when applying the findings we present in this paper. Given the definition above, examples of compliance constraints are laws imposed by a government, internal policies of an organization and contractual obligations with between organizations. In section 5 we present a crystallization of the properties of compliance regulations, based on a review of [3] and [8].

3 Background

To provide a technical context of CRLs, this section provides an introduction to their specifics. In this paper, we consider three CRLs, which will be described in detail to give some idea of their differences, applications and limitation. Two families of logic systems are represented in this paper, namely the *deontic* and *temporal* families. The prior distinguishes itself due to its concern with the expression of *obligations*, *prohibitions* and *permissions* [4]. Temporal logic on the other hand specifies propositions in terms of *time* [4]. This section concludes with a comparison the logical systems when expressing an identical regulation. The CRLs to be considered for our research are:

LTL This logic system allows for the formal specification of temporal properties for software and hardware [3]. LTL limits the range of states that can follow another state to one, representing them as linear state sequences [3]. Evidently, this allows only for the expression of a single process at the time in a system's behaviour. LTL is part of the family of temporal logic [3].

CTL Similar to LTL, CTL is used for the formal definition of temporal properties in the context of software and hardware design [3]. Additionally, CTL provides the means to have multiple *futures* — i.e. multiple states following on another – modelled as an infinite *computation tree*. This extends the expressiveness found in LTL with support for non-deterministic systems. Just like LTL, CTL is part of the temporal family of logic [3].

FCL While both LTL and CTL are part of the temporal logic family, FCL belongs to the family of deontic logic. As such, FCL distinguishes itself from the other two logic systems by expressing its propositions in terms of terms of *obligations*, *prohibitions* and *permissions*. LTL is particularly suited for expressing propositions that allow for some kind of violation (see also the property of *monotonicity* in section 5) [3].

Although this paper will not feature any instructions on the formulation of regulations in the aforementioned CRLs, an simple example of expressions is included in Table 1.

Representation	Expression
Textual	Only Post-processing Clerk and Supervisor roles can access the "Credit Bureau service".
LTL	$G(CheckCreditWorthiness.Role(Role1) \rightarrow G((Role1 = \text{'PostProcessingClerk'} \vee Role1 = \text{'Supervisor'}))$
CTL	$EG(CheckCreditWorthiness.Role(Role1) \rightarrow AG((Role1 = \text{'PostProcessingClerk'} \vee Role1 = \text{'Supervisor'}))$
FCL	$Role1 \neq \text{'PostProcessingClerk'} \wedge Role1 \neq \text{'Supervisor'} \vdash O_{Role1} \neq CheckCreditWorthiness$

Table 1. A simple example of different expressions of the same regulation, as found in [3].

4 Methodology

In order to present a framework that can capture the intrinsic features of compliance regulations and match them to the proper logical system, we will start our research by reviewing the properties implied in the works of [3], [4] and [8]. As no explicit definition of regulation properties have been found in existing literature, an extensive literature review must filter out any remarks on certain properties that are considered when applying a logical system. For example, in [3] the property of *fairness* is mentioned as follows:

> "Neither CTL nor FCL can express the weak fairness property of R5 (a constantly enabled event must occur infinitely often) [9], which is expressible in LTL. The same applies to the specification of strong fairness properties ."

This property of fairness must be considered when analyzing a constraint as it as an impact on the ability of the CRLs to express it properly. In this case, explicating the definition of fairness requires us to review the requirement in question and the mechanics of the CRLs that fail to capture this specific feature. Similar approaches have been utilized to extract the other properties in section 5, which have been encoded to allow easy referencing at a later stage.

The next step is reviewing the properties of logic systems. These are expresses as *requirements* in [3] and described in short. In section 6 we include this listing of requirements, extended with a more intuitive description to allow for an easier understanding of their meaning. These description also incorporate mentions of the properties of compliance constraints defined earlier to emphasize their importance and connectivity. Again, these requirements are encoded for later reference.

Combining our findings on both compliance constraints and logic systems, we will provide a *logical matrix* that indicates relations between compliance constraint properties and logic systems requirements. This tool will be the basis of our framework, as it can be used to reduce the amount of options (i.e. CRLs to pick from) by providing a subset of requirements to be considered.

In order to demonstrate the functionality of the tool we present, we conduct a case study. To this end, several constraints are adapted from [1] and [4]. We show how these constraints can be analyzed to identify their properties and how the critical requirements can be selected. Using the works of [3] we can then select the CRL that suites the use case best.

5 Proposing Properties for Compliance Regulations

Compliance regulations are usually expressed as a simple body of text, as exemplified in [3]. In this section, we will present the results of analyzing several compliance regulations presented in the literature. We compile our findings as a list of non-functional properties — as defined in section 2 — that allow us to judge the performance of the compliance constraint in a logic system. Several properties of compliance constraints can be identified in order to get a firmer grasp on its concept:

Source The document or entity from which the requirements originates, e.g. a governmental law, internal policy, etc. [3, 4]. This might be a-priory knowledge, as one might be aware of where a regulation has been obtained, or implied within the definition of the regulation itself.

Fairness Fairness indicates whether and when a process will be allowed to change state. Examples of values for this property are *strong* and *weak*, specifying the required method of requesting a change in state for a process [3]. This property appears in constraints when it specifies that one activity triggers another, as exemplified in constraint R5 in [3]:

> "If loan conditions are satisfied, the customer can check the status of her loan request infinitely often until the customer is notified."

Permission Constraints can assign permissions to actors. These are usually identifiable by words such as "allowed" and "rights". An example of a constraint containing the notion of permission is found in constraint R3 in [3]:

> "[T]he manager is notified by the system and Post-processing Clerk is allowed to do the check."

Redundancy The definition of a constraint might contain requirements or definitions that are irrelevant as they are either duplicates of other constraints or not applicable to the processes at hand [3].

Validity Specifies to which processes the constraint applies, e.g. *global validity* if a special process should be taken into account for the design of all processes or conversely *process specific* if it only applies to a single process [8].

Existence A regulation can require that something — e.g. activities, resources and roles – exists, either as a condition of something else existing or in general. For example, regulation R6 in [3] has this property:

> "[T]here exists an activity 'evaluation of the loan risk' that should be performed by the manager."

Control-flow Constraints are not limited to specifying activities that need to be performed, but can also require the order and timing of the activities [3]. Key words that are usually present in regulations are "prior", "after", "before", "until", etc.. In [6] we find an example of such a regulation:

> "All issues are covered prior to formulating a resolution."

Data validity Constraints might specify that data needs to be validated in general or when a condition has been met [3]. Words like "check", "verify" and "validate" might indicate such a property. An example of this is found in rule 1 in [6], which requires for data — in this case a resolution to a complaint — needs to be checked for acceptance by the customer:

> "Resolutions to complaints should always be checked for acceptance with the customer[.]"

Data requirements When data is being handled during business processes, it must adhere to certain requirements in order to be of use. Constraints can indicate these requirements [3]. Resolution Section 3 in [4] is an example of a constraint that determines a requirement for data, as it forbids collection of certain data types:

> "The collection of medical information is forbidden, unless acting on a court order authorising [*sic*] it."

Resource perspective A constraint might specify the rights of actors or tasks to be executed based on the existence of a certain resource. Looking at regulation R1 in [3] we can identify this property as it defines the roles required to access a specific resource:

> "Only Post-processing Clerk and Supervisor roles can access the 'Credit Bureau service'."

Monotonicity While most constraint are fairly strict, some can specify certain exception in which violation of the constraint is allowed to some or full extend — defined as non-monotonic [3]. Many examples of this exist in the related works, one of these being norm Section 3 in [4] as it explicitly states that the main regulatory constraint it defines can be overruled if a specific condition has been met:

> "The collection of medical information is forbidden, unless the entity [...] is permitted to collect personal information."

Real-time The time period available for the activities to be performed, which is an extra constraint on the activity [3]. In regulation R7 of [3] we find a very explicit constraint on the amount of time that should have elapsed before a certain activity is allowed to start:

> "The Credit Broker can start a loan [...] only if 5 workdays or more have elapsed since the loan approval form was sent."

Level Constraints can be written by people in any domain, resulting in different *views*, which also has an impact on the semantics [8]. Examples of different levels are *high level* i.e. written in natural language and *implementation level* i.e. written in language understandable by automated process management systems [8].

In Table 2 we present the specifications for the aforementioned properties of compliance constraints and their allowed value types and values. Using this table as a checklist, one can easily identify the properties of a regulation, assigning the proper value if appropriate. To this end, each property has been encoded for easy referencing.

6 Reviewing Requirements for CRLs

As per [3], several requirements can be defined to allow proper use of CRLs in a practical context. Some of these requirements can be linked to the properties of compliance regulation as defined in section 4. The requirements for CRLs as defined in [3] are:

Formality In order to allow automatic analysis, CRLs should be formulated with formal language. This is related to the *semantics* used to clarify the specifics of the constraint and the *level* of the definition, which must be implementation level i.e. very low.

Usability As the counterpart of formality, usability requires users (e.g. humans) to understand the constraint expressed in the CRLs. Highly complex constraints defined in a natural language already have a low usability, causing even lower usability when translated to a CRL. Having *semantics* that are hard to understand by users will lower the usability even more.

Expressiveness The expressiveness of a CRLs is related to the level of detail that it can capture. Too much expressiveness results in redundancy while too little will make the CRLs useless in for certain tasks. All properties of constraints are in some way related to expressiveness, as the language in which the constraint is formulated will determine whether or not the property can be expressed.

Declarativeness Constraint usually declare the activities that need to be started. Therefore, it is preferred that the CRL is of the declarative type. However, if a constraint has a high presence of *control-flow*, a declarative language will not be suitable, as the order in which the activities are executed is also relevant. This is due to the fact that a declarative system can only indicate which activities must be executed — which can be more than one — but not in what order.

Consistency checks Different constraints can have an effect on the same process and even cause conflicts. Therefore, it is desirable for a CRL to provide mechanisms that check whether or not the constraints are consistent with each other. Contradictions between constraints can be related to their property of *validity* — as a higher validity increases the chance of conflicts — and *monotonicity*.

Non-monotonicity A CRL should support the property of *monotonicity* of constraints – in particular its absence. This requires its syntax to express exceptions for certain situations to some extent.

Ref.	Property	Value
a	**Source**	Categorical value indicating the source of the constraint, such as an official document, internal policy, etc..
b	**Fairness**	Ordinal values ranging from *strong* to *weak*, indicating the required method for requesting a change of state.
c	**Permission**	Binary value indicating whether the constraint contains the notion of permission.
d	**Redundancy**	Binary value indicating whether the constraint contains redundant information.
e	**Validity**	Count value indicating the amount of processes for which the current constraint is valid or a value of $*$ indicating global validity.
f	**Existence**	Binary value indicating whether the constraint contains the notion of existence.
g	**Control-flow**	Binary value indicating whether the constraint requires the ordering and timing of activities.
h	**Data validity**	Binary value indicating whether the constraint specifies when an how data needs to be validated.
i	**Data requirements**	Binary value indicating whether the constraint specifies requirements for types of data.
j	**Resource perspective**	Binary value indicating whether the constraint specifies tasks or access rights for data.
k	**Monotonicity**	Binary value indicating whether the constraint allows violation in specific cases.
l	**Real-time**	Real value indicating the time period in hours, available for the activities to be performed or no value to indicate the absence of a time limit.
m	**Level**	Ordinal value indicating the level of the language in which the constraint is specified.

Table 2. The properties of compliance constraints and a proposed specification of the their allowed values.

Generic Constraints having the properties of *control-flow*, *data* and *resource-perspective* are very dissimilar, but can appear in the same use-case. In order to be able to use the same CRL for all constraints in such a use-case it should be general in the sense that it can express all of them well. This is related to the range of properties present in a use-case, as having a smaller range might allow for a CRL that is less generic and better suited for another property present in the constraints.

Symmetricity Having a symmetric CRL means that it allows for annotations for business process models that refer to compliance constraints. Using these annotations, the relations between constraints and processes can be inspected more easily.

Normalization The *redundancy* of a constraint is undesirable, as it lowers the usability and requires might require an unnecessary high level of expressiveness. A CRL that can normalize constraints will reduce the amount of redundancies.

Intelligible feedback Knowing that a constraint has been violated use usually not enough. Providing some kind of feedback on the cause of a violation and how to resolve it should be supported.

Real-time The constraint property of *real-time* should be supported, which specifies the time period within an activity must be performed.

In [3], the compliance of the aforementioned language to the requirements have been compared. The results of this comparison have been incorporated in Table 3 for convenience. A simplified visualization of the relations between the requirements and CRLs has been included in Figure 2.

Ref.	Requirement	FCL	LTL	CTL
1	**Formality**	+	+	+
2	**Usability**	−	−	−
3	**Expressiveness**	±	±	±
4	**Declarativeness**	+	+	+
5	**Consistency checks**	+	−	−
6	**Non-monotonicity**	+	±	−
7	**Generic**	±	±	±
8	**Symmetricity**	±	−	−
9	**Normalization**	+	−	−
10	**Intelligible feedback**	−	+	±
11	**Real-time**	+	+	+

Table 3. Comparison taken from [3] of the requirements met in the three CRLs under consideration in this paper, where + indicates full support, − indicates the lack of support and ± indicates partial support.

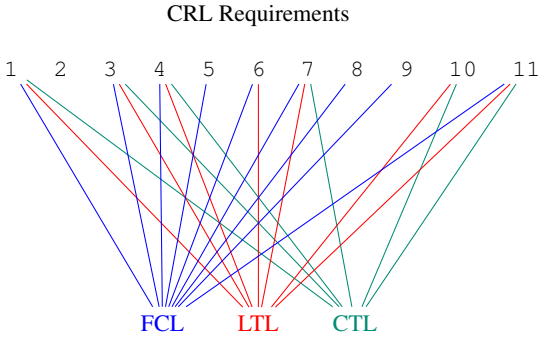


Fig. 2. A visualization of the relations between the CRL requirements and the CRLs considered in this paper: FCL, LTL and CTL.

7 Case Study: Analyzing Starfleet Regulations

In this paper, we aim to streamline the process of selecting the right CRL for per use-case. To this end, we present Table 4 in which we link the properties of constraints as presented in Table 2 to the requirement support of the CRLs included in Table 3. With this table, we provide a tool that can be used the reduce the amount of requirements based on properties present in a use-case. In Figure 3 we provide a visualization of the relations between the requirement properties and CRL requirements. In order to facilitate a demonstration of the usage of the analyzing tool presented in Table 4, a small set of constraints have been defined in Table 5.

Looking at constraint i Table 5, we can identify the presence of the *resource perspective* property. Similarly, constraint ii restricts employees to access specific data during an activity but has a higher *validity*, as it segregates two processes. In constraint iii we find *control-flow*, as several activities are ordered. Furthermore, we find *fairness*, as the constraint specifies an activity as the result of another activity. Constraint iv is similar to iii, as it also specifies to moment at which

	1	2	3	4	5	6	7	8	9	10	11
a	1	1	0	0	0	0	0	0	1	0	0
b	0	0	1	0	0	0	0	0	0	0	1
c	0	0	1	1	0	0	0	0	0	0	0
d	0	0	1	0	0	0	0	0	1	0	0
e	0	0	1	0	1	0	0	0	0	1	0
f	0	0	1	1	0	0	0	0	0	0	0
g	0	0	1	1	0	0	1	0	0	0	0
h	0	0	1	1	0	0	1	0	0	0	0
i	0	0	1	1	0	0	1	0	0	0	0
j	0	0	1	1	0	0	1	0	0	0	0
k	0	0	1	0	1	1	0	0	0	0	0
l	0	0	1	0	0	0	0	0	0	0	1
m	1	1	1	0	0	0	0	1	0	0	0

Table 4. A binary matrix visualizing the relations between the properties of compliance constraints and the requirements of CRLs, where a black cell indicates the presence of such a relation. The properties in the rows and the requirements in the columns are referenced according to their encoding in Table 5 and Table 3 respectively. A relation of any kind is indicated by a value of 1 and a black cell background, while the lack of such a relation is marked by a value of 0.

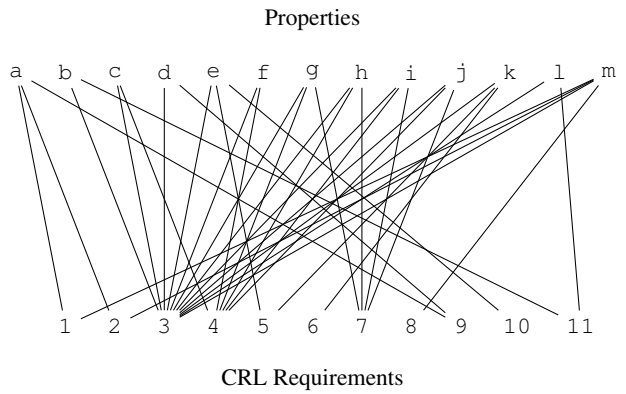


Fig. 3. A visualization of the relations between the regulation properties and CRL requirements, as represented as a binary matrix in Table 4.

activities must be executed. Similar to constraint i and ii, constraint v specifies access rights to data, but does so for the customer instead of the employee. Constraint vii has the property of *real-time*, as it explicitly states an amount of time required to pass before an activity can be started. Next, constraint viii is interesting as it allows for violation in specific cases, and is therefore *non-monotonic*. Finally, constraint ix specifies the *validity* of data is specific cases. A more detailed analysis of the use-case is include in Table 6. Looking at the results in Table 6 we can observe that the property of *redundancy* is irrelevant for this particular use-case as it has a value of 0 for all the constraints. Therefore we can conclude that the relevant properties in this case are *fairness* (b), *permission* (c), *validity* (e), *existence* (f), *control-flow* (g), *data requirements* (i), *resource perspective* (j), *monotonicity* (k) and *real-time* (l). Cross-referencing these properties with the requirements for CRLs using Table 4 we can determine that the relevant requirements are *expressiveness* (3), *declarativeness* (4), *consistency checks* (5) and *real-time* (11). Ignoring the requirements for which the performance is equal for all CRLs, we are left with the requirement of *consistency checks* to determine the most suitable language to define our use-case in — which is FCL.

Ref.	Source	Constraint
i	IP	Only engineers and architects can access the blueprint database.
ii	UFP	Customer space travel permit check is segregated from credit worthiness check.
iii	IP, UFP	If the order of the customer exceeds 1 trillion Federation credits his or her credit worthiness must be checked by the sales supervisor immediately. In case of absence of the supervisor a suspense file is created. In case of failure of the creation of a suspense file, the manager is notified by the system and a lower ranking sales person is allowed to do the check.
iv	UFP	As a final control, the branch office manager has to check whether the ordered space craft is not a potential risk to intergalactic peace.
v	IP	If an order is accepted, the customer can check the construction status of the spaceship infinitely often until it has been delivered.
vi	IP	If a customer space travel permit check is performed and the order includes a warp drive, then there exists an activity "warp drive eligibility check" that should be performed by the manager.
vii	UFP	The lead engineer can start realization of the space ship design (approved by the customer), only if 5 workdays or more have elapsed since the spacecraft approval form was sent.
viii	UFP	The collection of personal information of customers is forbidden unless required to meet other regulations by the UFP.
ix	UFP	The destruction of illegally collected personal information before accessing it is a defence against the illegal collection of personal information of customers.

Table 5. A collection of constraints for a fictional spaceship construction company *Starfleet Shipyards*, provided to demonstrate the usage of several logical languages. The compliance constraints have been defined in natural language and are adapted from examples found in [3] and [4]. The (also fictional) sources of the constraints are either the internal policy (IP), regulations enforced by the United Federation of Planets (UFP) or a combination thereof.

	b	c	e	f	g	i	j	k	l
i	0	1	1	0	0	0	1	0	0
ii	0	0	2	0	0	0	0	0	0
iii	1	0	1	0	1	0	0	1	0
iv	0	0	1	0	1	0	0	0	0
v	1	0	1	0	1	0	0	0	0
vi	1	0	1	1	0	0	0	0	0
vii	0	0	1	0	1	0	0	0	120
viii	0	0	*	0	0	1	0	1	0
ix	0	0	1	0	0	1	0	0	0

Table 6. An overview of the values of a subset of the properties of the constraints as defined in Table 2 and Table 5 respectively, omitting those not present in any of the constraints.

8 Discussion on Analysis and Work Flow

The demonstrated method of analyzing regulations allows for a formal definition of a work flow, describing how to express textual regulations using a CRL. The product of the analysis of the textual constraints yields the constraints expressed as collections of properties and their values. Using these properties, the requirements for the CRL can be

identified, ultimately allowing for a selection of the most appropriate CRL. Combining the selected CRL and the textual representations, the regulations can be expressed such that they can be processed using an automated compliance regulations verification system. This work flow has been visualized using a simple flowchart in Figure 4.

To improve the process of analyzing the properties of regulations, certain tools might be utilized, such as Microsoft Excel or similar computer programs. This also allows for recording changes in properties when regulations changes in definition, which might lead to the identification of erroneous definitions or changes.

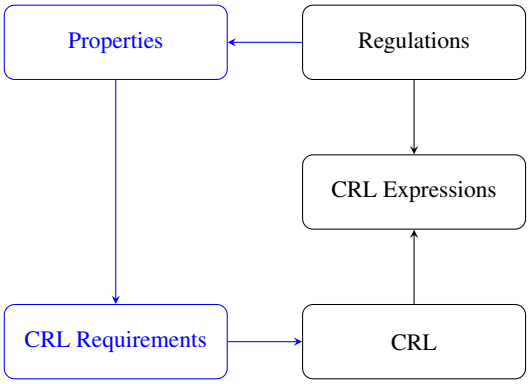


Fig. 4. A simple flowchart representing the proposed work flow for expressing textual regulations with a CRL, using the analysis of regulation properties. Recall the flowchart in Figure 1, for which the the dotted relation can be substituted with our proposed process, highlighted in this graphic.

9 Conclusion and Outlook

In this paper, we have provided the means to analyze a set of regulations, streamlining the process of expressing them in a CRL, as defined in section 8 and illustrated in Figure 4. This fills a gap in the complete process of implementing compliance regulation verification systems as defined in section 1 and visualized in Figure 1. Using our methodical approach, the effectiveness of a CRL can be validated, making sure that there are no unexpected shortcomings to the logical system. As a result, expressing compliance regulations using a CRL allows business analysts to develop a robuster verification system for their business processes. Further research is needed to verify the completeness of our proposed definition of regulation properties, as the set of regulations that have been analyzed during our research might be too limited to include all variations. Furthermore, other logical systems than the three mentioned in this document can be analyzed in order to identify their compliance to the requirements to extend the proposed work flow to a wider range of methods and techniques.

Finally, an interesting subject of research might be a review of tools that can be utilized in order to analyze regulations. Ultimately, this might lead to a compilation of requirements for such a tool, which can become the foundation of the development of more specialized utilities.

Acknowledgements

The authors wish to thank dr. Heerko Groefsema for his expertise.

References

[1] Decker G. Weske M. Awad, A. Efficient compliance checking using bpmn-q and temporal logic. *BPM'08*, 2008.

[2] Lianping Chen, Muhammad Ali Babar, and Bashar Nuseibeh. Characterizing architecturally significant requirements. *IEEE Software*, 30(2):3845, 2013. doi: 10.1109/ms.2012.174.

[3] Amal Elgammal, Oktay Turetken, Willem-Jan van den Heuvel, and Mike Papazoglou. On the formal specification of regulatory compliance: A comparative analysis. *Service-Oriented Computing Lecture Notes in Computer Science*, page 2738, 2011. doi: 10.1007/978-3-642-19394-1_4.

[4] Guido Governatori and Mustafa Hashmi. No time for compliance. *2015 IEEE 19th International Enterprise Distributed Object Computing Conference*, 2015. doi: 10.1109/edoc.2015.12.

[5] Guido Governatori, Zoran Milosevic, and Shazia Sadiq. Compliance checking between business processes and business contracts. In *2006 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC06)*, page 221232. IEEE, October 2006. doi: 10.1109/edoc.2006.22.

[6] Heerko Groefsema, Nick van Beest, and Marco Aiello. A formal model for compliance verification of service compositions. *IEEE Transactions on Services Computing*, 11(3):466479, 2016. doi: 10.1109/tsc.2016.2579621.

[7] Mathias Kirchmer. *High Performance Through Business Process Management Strategy Execution in a Digital World*. Springer International Publishing, 2017.

[8] Linh Thao Ly, Kevin Göser, Stefanie Rinderle-Ma, and Peter Dadam. Compliance of semantic constraints — a requirements analysis for process management systems. In Shazia Sadiq, Marta Indulska, Michael zur Muehlen, Xavier Franch, Ela Hunt, and Remi Coletta, editors, *Proceedings of the 1st International Workshop on Governance, Risk and Compliance: Applications in Information Systems*, volume 339 of *CEUR Workshop Proceedings*, pages 31–45, Montpelier, June 2008. CEUR Workshop Proceedings.

[9] Amir Pnueli. The temporal logic of programs. *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, page 4657, 1977. doi: 10.1109/sfcs.1977.32.

[10] Wil M. P. van der Aalst and Maja Pesic. A declarative approach for flexible businessprocesses management. In *Business Process Management Workshops Lecture Notes in Computer Science*, pages 169–180. Springer, 2006. doi: 10.1007/11837862_18.

Distributed Constraint Optimization: A Comparison of Recently Proposed Complete Algorithms

Sofie Lövdal, Elisa Oostwal

Abstract— Constraint optimization problems (COPs) is a class of problems in which the variable assignments need to adhere to a number of constraints. The goal of a COP is to find the set of assignments that optimizes an objective function. This is known to be an NP-hard problem, which makes it difficult to solve problems having a large number of variables or constraints. One approach to solving a COP is using a multi-agent system: the variables and constraints are distributed among the agents, effectively splitting the global COP into smaller (local) COPs. The agents then communicate their state in order to find a solution. Since the agents in such a system operate autonomously, defining a model for communication between the agents is a non-trivial task. Distributed constraint optimization (DCOP) is an active research field in artificial intelligence which provides a model for the needed coordination and distributed problem solving. Algorithms for solving distributed constraint optimization problems (DCOPs) can be divided roughly into two categories: complete and incomplete algorithms. In this paper we restrict ourselves to the former set of algorithms, where current research aims to improve the existing complete methods. We give an overview of the current state of the field and investigate which algorithms perform best under which conditions. Seven algorithms with different solving strategies are reviewed and compared, where special attention is given to some state-of-the-art methods that have been proposed over the recent years. The algorithms are compared based on their efficiency, which is determined by their spatial complexity and communicative properties. Their applicability is also considered, since the performance of a DCOP algorithm depends heavily on the characteristics of the problem at hand. In the end we have created a guideline for selecting the best suitable algorithm based on the type of problem, which is visualized by a decision-tree. We conclude that ConcFB outperforms many of the algorithms, making it the most promising complete DCOP algorithm. We note, however, that it is not the most suitable choice for all types of problems. We therefore encourage scientists to use our guideline for selecting the algorithm that is best suitable for their problem.

Index Terms—Constraint optimization problems, Constraint satisfaction problems, Distributed constraint optimization problems, Complete DCOP algorithms, Multi-agent systems.

1 Introduction

Multi-agent systems is an important field in Artificial Intelligence. It can model a large class of real-world problems by allowing multiple agents to interact in an environment. In particular, agents can be assigned a task for which a global objective function needs to be optimized, or for which resources are limited. These types of environments pose a number of constraints that need to be satisfied. The agents then need to cooperate in order to reach the common goal, and should thus communicate their decisions. Since the communication between agents defines the quality of the outcome, the need for proper coordination models is urgent. However, since each agent is its own entity and takes actions independently, coordinating the agents' actions is a non-trivial task.

Distributed constraint optimization problems (DCOPs) have risen as a way of modeling this coordination, which allows for problem solving. The problem is divided into smaller subproblems, which are then distributed among the agents. The agents operate in their local environment in an attempt to solve their subproblem, while ensuring that the reward is maximized or the cost is minimized. At the same time, the restrictions on the global environment still need to be respected. DCOP algorithms provide methods for communication between agents in such a way that they can obtain the best possible outcome given the full set of constraints. In particular, the set of complete algorithms will guarantee an optimal solution to the problem.

While the best choice of DCOP algorithm depends on the properties of a given problem, recent literature mostly focuses on presenting new methods or comparing the performance of DCOP algorithms on a single benchmark problem. Therefore, there is a need for research providing guidelines on which type of algorithm is suitable for what kinds of problems. In this paper we explore several types of complete DCOP algorithms by defining a taxonomy. We review and discuss algorithms from each category, assessing the algorithms based on their applicability and efficiency. We have summarized these results in a guideline, which gives recommendations on choice of algorithm with regards to the problem structure and available computational resources.

First, in section 2, we describe the concepts related to DCOP which are needed to understand the algorithms. We also present a taxonomy of complete DCOP algorithms. For each node of the taxonomy tree, corresponding to a category, we pick an algorithm. We explain the selected algorithms and briefly describe their advantages and disadvantages in section 3. They are then compared to one another in the section that follows, section 4. Based on their properties we created a guideline which can be used to pick the most suitable algorithm for a given problem. Finally, we summarize our findings in section 5.

- *Sofie Lövdal is with University of Groningen, E-mail: s.s.lovdal@student.rug.nl.*
- *Elisa Oostwal is with University of Groningen, E-mail: e.c.oostwal@student.rug.nl.*

2 Background

In this section, definitions are presented of concepts that form the basis of the DCOP research field. We categorize the algorithms and describe the differences between the categories. We also explain how DCOPs are represented, and what communication schemes can be used based on these representations. All of this information is needed to understand the DCOP algorithms presented in section 3.

2.1 Multi-agent systems

An intelligent agent is an entity which is capable of collecting data about its environment using sensors. It can act upon the perceived information and any information it has already stored [1]. The agent decides on its next action by considering the impact it has on its objective function or cost function. Multiple intelligent agents can cooperate in order to maximize their performance, which is based on a common goal. Such a setting is referred to as a multi-agent system. An example of a multi-agent system would be autonomous vehicles. In this context a car is considered to be an agent that needs to take actions based on the environment its sensors can observe. The car-agents all

want to optimize their own objective function of reaching their destination as fast as possible. At the same time, accidents, traffic jams, and speed limit violations should be avoided. Such a traffic flow requires a sophisticated coordination scheme between the individual agents of the system.

2.2 Distributed constraint optimization problems

One method for coordinating a multi-agent system is modeling it as a DCOP. Here, we solve problems in which constraints are put on variables, which each have a finite and discrete domain.

One class of problems that can be solved using this approach is constraint satisfaction problems (CSPs). Here, a combination of variable assignments is sought such that all constraints are satisfied. A solution can only be satisfactory or unsatisfactory: either it satisfies all constraints or it does not. In a Distributed CSP (DisCSP) the variables and constraints are distributed among agents, creating local CSPs at every agent. Each agent tries to solve their local CSP in order to find a satisfactory solution to the global CSP [2].

A similar class of problems exist in which constraints are modelled by cost functions or reward functions, rather than merely being satisfied or not. The aim is to find a solution which optimizes the cost function of the problem. As a result of this difference, the methods that can be applied to CSPs can, in general, not be used to solve these constraint optimization problems (COPs). These problems gave rise to the field of distributed constraint optimization problems (DCOP), where the variables, and constraints associated with them, are distributed among agents.

Formally, a COP consists of a set of variables $V = \{v_1, v_2, ..., v_n\}$, where each variable v_i has a finite and discrete domain D_i, and a set of constraints C between the variables. Each constraint $c_{ij} \in C$ defines the cost for an assignment to two variables v_i and v_j with regards to any value from their domains. Formally, $c_{ij} : D_i \times D_j \rightarrow R^+$ [3]. Even though one variable can be connected to many other variables by means of constraints, all constraints can be modeled as binary.

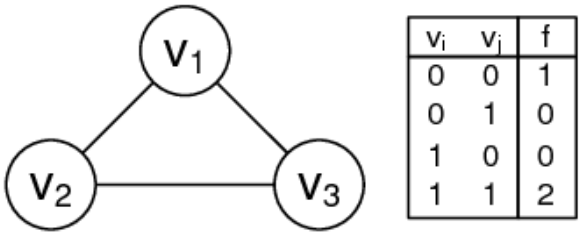
A DCOP consists of a set of agents $\{A_1, A_2, ..., A_k\}$, a global COP, and a set of local COPs $\{P_1, P_2, ..., P_k\}$, where P_j is a subset of variables and constraints of the global COP. Agent A_j is the only agent that can assign values to the variables of P_j; the other agents do not have access to the variables belonging to P_j [2]. Even though the agents only have a limited view of the global environment, they should be able to perform actions that support optimizing the global objective function that has been defined for the problem [4].

Let us consider the example DCOP in Figure 1, where each variable is assigned to an agent. The aim is to maximize the reward function, defined to be the sum over the rewards given to variable assignments. On the left part of the figure, the variables and their constraints are shown as a constraint graph. A node corresponds to a variable, a connection between two variables represents a constraint between the two variables. On the right a table is presented describing the cost accompanied with value assignments. The table can be interpreted as such: if we assign the value 0 to both variable v_1 and v_2, this has a reward of 1. However, if we assign the value 1 to both variables this has a reward of 2. Hence, $\{v_1, v_2\} = \{1, 1\}$ is the assignment which results in the optimal reward. Similarly, we can reason that v_3 should also have the value 1. In conclusion, the set of assignments to $\{v_1, v_2, v_3\}$ that maximizes the reward is $\{1, 1, 1\}$, with a total reward of 6.

2.3 DCOP algorithms

Several algorithms have been developed which are capable of solving DCOPs. They can be split into two main categories: complete algorithms and incomplete algorithms. The former group assures convergence to an optimal solution, which is useful when solution quality needs to be guaranteed. As a drawback, however, they require significantly more computational resources. The effort for computation and communication grows exponentially with the number of agents in the system [4].

On the other hand, methods from the latter group restrict the search space, which limits the memory usage as well as execution time. This makes incomplete algorithms suitable for large-scale problems and



v_i	v_j	f
0	0	1
0	1	0
1	0	0
1	1	2

Fig. 1: Example of a DCOP. On the left: the problem is displayed as a constraint graph. On the right: a table describing value assignments and their associated cost. The set of assignments that maximizes the reward is $\{v_1, v_2, v_3\} = \{1, 1, 1\}$, with a total reward of 6 [5].

real-time applications. We note though that as a result of restricting the search space, the algorithm can only guarantee a lower bound with regards to the solution quality.

In this paper we restrict ourselves to the set of complete algorithms.

2.3.1 Taxonomy

The algorithms can be categorized even further, based on their solving strategy and model of communication. Yeoh et al. proposed a taxonomy which provides a good overview of the main features of each group of algorithms [6]. Leite et al. introduced a variation on this, which we have used as a guide in our selection of algorithms covered in this article [4]. The taxonomy applied in this paper is shown in Figure 2. The taxonomy divides the algorithms into categories based on their structural properties. In the following sections we describe the differences between the categories.

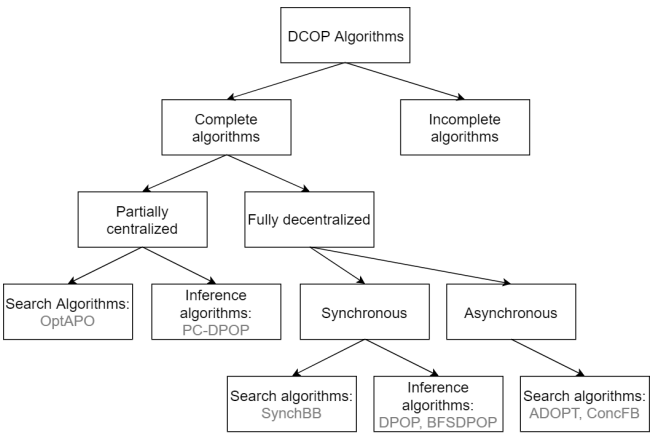


Fig. 2: Categories of DCOP algorithms. The node 'Incomplete algorithms' has not been categorized since it is not treated in this article. The figure has been adapted from the taxonomy suggested in [4].

2.3.2 Fully centralized vs partially centralized

A DCOP algorithm can have different levels of centralization, which refers to the level of information sharing between the agents. This is also connected to how agents solve internal conflicts. Fully decentralized algorithms let agents take a local decision without communicating with other agents. However, this leads to more communication and computation steps later on in the process, since conflicts originating from earlier steps need to be resolved. This communication structure provides for a minimal loss of information privacy.

Partially centralized algorithms let the agents take a local decision before informing its peers involved in the same sub-problem. This decreases the risk of getting stuck at local maximums and reduces

conflicts. However, it also causes partial loss of information privacy, which we usually want to avoid when modeling a multi-agent system.

2.3.3 Synchronous vs asynchronous

Besides having a degree of centralization, algorithms can operate either synchronously or asynchronously. The agents of asynchronous algorithms operate independently, completely at their own pace, relying only on their local view of the environment. Communication still occurs between agents, but they do not wait for incoming messages before performing an assignment. Although this reduces the idle-time of agents, it may also lead to inconsistent views of the environment, which increases the risk of conflicts. On the contrary, fully synchronously operating algorithms have a systematic ordering defined. The agents need to wait for their peers to communicate specific messages before they are allowed to take action. The view of the environment is consistent, but it also causes a significant amount of idle time for the agents.

2.3.4 Inference vs search algorithms

The leaf nodes in our taxonomy are labelled as either inference algorithms or search algorithms, according to their strategy for state space exploration (refer to Figure 2). Search-based algorithms make use of common methods such as best-first search or depth-first search (DFS) before using backtracking and the branch-and-bound algorithm to go through the state space. Inference based algorithms let agents consider the accumulated constraint costs of peers in its local environment. This information is then shared between neighbors in order to reduce the problem size locally.

2.4 DCOP representation and communication structures

A DCOP can be represented as a graph, where the nodes represent variables and the edges represent the constraints between pairs of variables (refer to Figure 3a). By ordering the constraint graph, a communication structure can be derived from it, which can then be used in DCOP algorithms. Three types of communication structures exist: original graph, chain structure, and pseudo-tree structure (refer to Figure 3b). An original graph is commonly used in incomplete algorithms, while chain- and pseudo-tree structures are frequently used in complete algorithms [7].

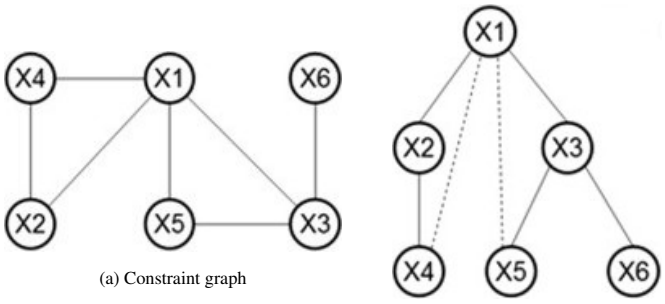
A chain structure forces agents to execute in sequence, which leads to poor efficiency. As an alternative, a pseudo-tree structure was proposed which enables agents in different branches to operate concurrently. This is possible due to the relative independence of nodes located in different branches [8]. To date this is the most commonly used communication structure [7].

A pseudo-tree is used when agents need to be ordered by priority. The tree is constructed from the constraint graph by using a search algorithm. We note however that the quality of a pseudo-tree, and therefore the performance of an algorithm using the pseudo-tree, depends on the search strategy used to construct it. While depth-first search (DFS) is frequently used, recently Chen et al. showed the benefits of using breadth-first search (BFS) [7]. The proposed algorithm will be covered in section 3.5.

Formally, a pseudo-tree is a spanning tree whose nodes may be connected with other higher priority nodes. One of the higher priority node is defined as parents. The other higher priority nodes are then referred to as pseudo-parents. The nodes down the tree which are connected to a pseudo-parent are labeled pseudo-children [4].

3 Complete DCOP algorithms

Complete DCOP algorithms are algorithms that guarantee obtaining an optimal solution to the constraint optimization problem at hand. In the following we present algorithms that are representatives of the leaf nodes in our taxonomy of complete DCOP algorithms (see Figure 2). We have chosen this setup in order to cover the full tree of the taxonomy. The algorithms presented have been chosen based on their relevance, which has been acknowledged in the literature, and reported performance in different scenarios.



(a) Constraint graph

(b) Pseudo-tree

Fig. 3: A DCOP represented as a graph. Variables are depicted as nodes, and constraints between two variables are depicted as edges. The constraint graph in (a) has been transformed into a pseudo-tree in (b). In a constraint graph all constraints are expressed as direct connections, while in a pseudo-tree the agents owning each variable are ordered. A constraint can then be directly or indirectly represented by the pseudo-tree hierarchy.

3.1 SynchBB

Introduced in 1997, the Synchronous Branch and Bound algorithm (SynchBB) was the first complete DCOP algorithm [4]. It is a decentralized, fully synchronous algorithm. It makes use of the Branch and Bound algorithm for solving combinatorial optimization problems and adapts it for a distributed environment. It requires agent and value ordering to be known and constant. The agents take turns of sequentially passing a partial assignment for all variables, so called path, to the next agent in line [9]. Initially, only one variable has an assignment. Every agent evaluates the path received and the first value of its domain in value ordering, and passes this on to the next agent if the agent evaluates it as less than the current upper bound. If it is more than the upper bound, it takes the next value in the value ordering domain. If none of the values in the domain suffice, the agent returns the path to the previous agent to apply backtracking. These steps are then applied recursively if necessary. Since the agents are operating sequentially and the variable and value ordering is fixed, SynchBB enables agents to do an exhaustive search in distributed search spaces. This means that the algorithm indeed finds a solution if one exists. A drawback is, however, that it cannot be extensively parallelized. This is due to that the agents inherently need to operate sequentially.

3.2 OptAPO

While in most DCOP algorithms the agents only have knowledge about their local COP, in partially centralized algorithms like Optimal Asynchronous Partial Overlay (OptAPO) knowledge about the other agents is partially shared [10]. The key idea in OptAPO is cooperative mediation, where a dynamically chosen agent is chosen as mediator for a subproblem.

In OptAPO, agents are able to share their knowledge to improve their local decisions. When an agent acts as a mediator, a solution is computed for a fraction of the global problem, and new assignments are recommended to the agents involved in the mediation session. The size of the subproblems increases over the course of the algorithm. This strategy reduces the cost of communication which would normally be required in fully centralized algorithms.

Each agent holds two lists: the agent-view and the good-list. The former contains information about connected agents and their names, values, domains and constraints. The good-list contains names of agents that share direct or indirect constraints with the agent in question. The agent with the largest good-list involved in a mediation session is chosen as the mediator, since this agent has the most knowledge about the relevant variable.

3.3 DPOP

Distributed Pseudo-tree Optimization Procedure (DPOP), introduced by Petcu et al, uses a pseudo-tree as its communication structure [11]. Rather than using a search strategy such as DFS to traverse the tree and solve the problem, the algorithm is based on inference. The idea is to compute and propagate the total cost of the variables.

The algorithm consists of three phases. In the first phase the agents transform the constraint graph into a pseudo-tree using DFS.

In the second (UTIL) phase, the agents compute the cost of all assignments considering its neighboring agents. The agents initiate this process, referred to as utility propagation, by sending UTIL messages to the parent of the pseudo-tree. A UTIL message contains the cost for each set of assignments from the sub-problem started in the sender agent [4]. These messages then propagate down the tree until they reach the leaf nodes, where the cost can be instantly computed by inspecting the constraints with their immediate neighbors [11]. These costs are then propagated back up in the direction of the parent, who ultimately collects the costs of all its children.

When the root node of the pseudo-tree has received all UTIL messages, the third (VALUE) phase is started. The root node chooses an assignment that optimizes the global cost of the problem. It then sends out its decision using VALUE messages, which then propagate to its children. Upon receiving the message from its parent, the child node becomes an active agent which chooses an assignment that optimizes the global cost function. In turn, it propagates the VALUE message to its children, including the assignment which it has chosen. This process continues until the message reaches the leaf nodes, which make the final assignment.

Although DPOP only requires a linear number of messages to solve a DCOP, the message size grows exponentially because of the propagation process [4, 7]. Many variants of DPOP have been proposed which try to tackle this problem. One of the earlier attempts is the Partially Centralized DPOP (PC-DPOP) algorithm, which was developed by Petcu et al., who originally proposed the DPOP algorithm [12]. Recently Chen et al. proposed a new algorithm which makes use of BFS to construct the pseudo-tree [7]. Their BFSDPOP algorithm effectively deals with the problem of message size and, in addition, enhances parallelism.

3.4 PC-DPOP

To overcome the space problems of DPOP, Petcu et al. introduce a control parameter k that puts bounds on the message dimensionality [12]. This allows for predictions about the computational requirements. Vice versa, the control parameter k can be adjusted to meet the hardware specifications of the system that is solving the problem.

The algorithm proceeds as normal DPOP, except in the UTIL phase. Like in DPOP, the leaf nodes start the utility propagation back up the tree. However, as soon as the outgoing UTIL message of a node has more than k dimensions, centralization begins. Instead of computing its UTIL message and propagating it to its parent, the node sends a Relation message to its parent. This message contains the set of relations that the node would have used as an input for computing the UTIL message. The parent node then becomes a cluster root which reconstructs the subproblem from the incoming Relation messages and solves it in a centralized fashion using an algorithm of its choice [13]. It adds its assignments to the UTIL message and then continues the UTIL propagation as in DPOP. The partial centralization ensures a maximal message size which is exponential in k, which performs better than DPOP in cases where the tree is dense. The partial centralization has as a drawback however that there is partial loss of privacy. The authors mention though that the privacy loss can be predicted and therefore restricted beforehand.

3.5 BFSDPOP

As mentioned before, the pseudo-tree used in DPOP is constructed using DFS. Chen et al. propose to use BFS instead, since a tree constructed with this strategy has more branches, which aids parallelism, as well as a lower height, which shortens the communication path and hence communication time [8]. They therefore argue that using BFS as strategy for producing the pseudo-tree allows for better parallelism and results in a higher communication efficiency [7]. Moreover, they manage to slightly reduce the message size, although it still grows exponentially. The largest difference is that in DFS the message size is exponential in the width of the tree, while in BFS the size is exponential with the number of cross-edges in the tree. They do, however, propose a method to limit the number of cross-edges, which restricts the message size. A more extensive comparison is made in section 4.1.

3.6 ADOPT

Asynchronous Distributed Optimization (ADOPT) makes use of fully asynchronously operating actors in order to solve the problem. It was the first fully decentralized, asynchronous search algorithm that was able to find an optimal solution [14].

ADOPT requires the agents to have an internal ordering based on a pseudo-tree. Each agent continuously considers its local environment and based on that takes the action it considers most optimal in a best-first manner. Each agent also keeps values for the current lower bound of the solution, as well as an upper bound. These values describe the minimal (sum of the cost of the constraints that is consistent with the current context) and maximal (best so far) values of the cost of the solution at the current state of the search. The upper bound is initialized as infinity, and the lower bound as the sum of the costs of the constraints between the (connected) ancestors [15]. The lower bound is iteratively updated as new cost information is communicated by the children of an agent. The algorithm has found its solution when these two bounds become equal for the agent in the root node of the pseudo tree.

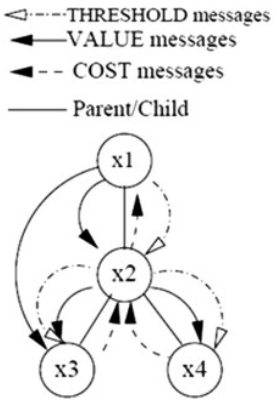


Fig. 4: Adopt communication model [14]. VALUE messages containing an assignment to a variable are sent from parents to children, and the children report back the cost of this assignment in a COST message. THRESHOLD messages are used to control backtracking and give information on the bound on the optimal cost of a solution.

There are three types of messages between agents in ADOPT (see Figure 4). If a parent node changes the value for a variable, it communicates this to its child nodes. The child verifies whether this value is in agreement with its local view, and if not, assigns another value to the variable to minimize the cost of the solution. The children send cost messages to its parents, containing the cost of its local view and the current lower and upper bound. Parents also send messages to children to update their threshold variable. The threshold controls the backtracking mechanism of the algorithm, informing the child nodes not to try to search for a solution with a cost lower than some lower bound that has been determined based on previous experience by a parent node.

3.7 ConcFB

Concurrent Forward Bounding (ConcFB) is an algorithm proposed by Netzer et al. which combines multiple processes of concurrent search and synchronized forward bounding [3]. The former provides for rapid exploration of the state space, and the latter provides possibilities for

early pruning: aborting the exploration of a solution when it displays weak potential. The search processes run on all agents concurrently and at the same time explore non-intersecting parts of the search space. This is done by dividing values of possible assignments to variables across search processes. The exploration of the search space itself is then done by synchronized forward bounding.

The agents are globally ordered within each search process. Every search process also maintains a current partial assignment (CPA) structure, containing the values that have been assigned to variables within that specific search process. The search processes share their best upper bound across all parts of the global search space. If a new, lower value on the best upper bound is communicated that particular search space can be pruned. New search processes are spawned dynamically by the algorithm if it detects new search spaces that seem promising. Finally, ConcFB will have explored every possible combination of assignments to the set of variables excluding the pruned options. The agent initiating the algorithm will then hold the value of the optimal solution, having collected results from the search processes it spawned. This will eventually be equal to the best upper bound.

4 Discussion

In this section, we compare and evaluate the algorithms explained in section 3. Two aspects are treated: the efficiency of the algorithms and their applicability. The former can be expressed analytically, while the latter will be expressed qualitatively, namely by the type of the problem or properties of the problem for which the algorithm is appropriate. Efficiency has been evaluated by analyzing the memory usage, the number of messages required by the algorithms, and the size of these messages. The runtime of the algorithms has not been taken into account because it is exponential for every complete DCOP algorithm [16]. The applicability of each algorithm depends on the structure of the problem at hand. This allowed us to create a guideline that summarizes our findings with regards to the applicability. In this guideline we have also taken efficiency into account in order to give the best recommendation.

4.1 Efficiency

In the following we describe the advantages and disadvantages of the algorithms treated in our paper, considering their efficiency. As stated above, the efficiency of a complete DCOP algorithm depends on the number of messages and their individual size (communication complexity), as well as on the memory required (space complexity). We have summarized the space and communication complexity of the algorithms treated in our work in Table 1, which is an adaptation of the tables presented by Petcu et al. [13] and Fioretto et al. [17], respectively. Here, it can be especially noted that no single algorithm displays a significantly lower total complexity than another - at least one of the categories is always exponential in complexity.

Table 1: Space and communication complexity of the covered algorithms. The table has been adapted from [13] and [17].

Algorithm	Number of messages	Message size	Memory
OptAPO	Exponential	Linear	Polynomial
ADOP	Exponential	Linear	Polynomial
SyncBB	Exponential	Linear	Linear
ConcFB	Exponential	Linear	Linear
DPOP	Linear	Exponential	Exponential
BFSDPOP	Linear	Exponential	Exponential
PC-DPOP	Linear	Exponential	Polynomial

SynchBB is the oldest and simplest DCOP algorithm. However, due to its fully synchronous processing with a pre-defined ordering of agents, a considerable amount of the computational resources are wasted since the agent need to wait for its turn a large amount of the time [3]. Its advantage lies however in its simplicity, making it very suitable for educational purposes. Especially if the problem size is small, SynchBB might be one of the most straightforward options for a complete DCOP.

It has been shown that the partially centralized communication pattern of OptAPO requires significantly less communication than ADOPT, even though they both are worst-case exponential in terms of the complexity of number of messages (refer to Table 1). However, OptAPO has scaling problems as the problems grow dense, since several mediators often solve overlapping problems [18].

Compared to OptAPO, PC-DPOP provides better control over what parts of the problems are centralized and allows this centralization to be optimal with respect to the chosen communication structure [13]. Based on experiments, Petcu et al. conclude that it also has strong efficiency gains over OptAPO. Although partially centralized inevitably comes with loss of privacy, PC-DPOP has ways of predicting privacy loss, which enables it to limit the loss. Additionally, its setup is such that it restricts the message size and memory usage.

For inference-based algorithms, the temporal and space complexities of the messages both depend on the maximal message size [7]. DPOP requires a linear number of messages to solve a DCOP [11]. Furthermore, A VALUE message contains at most a linear amount of assignments, while a UTIL message can contain an exponential number of values. Therefore, for DPOP, the temporal and space complexities of disposing a message both lie on the maximal UTIL message size, which is exponential in the width of the DFS pseudo-tree.

The BFSDPOP algorithm requires the same number of messages as DPOP, and hence has a linear complexity in the number of messages. Similarly to DPOP, the temporal and space complexities of communication also depend on the maximal UTIL message size. Chen et al. [7] prove in their article that the largest UTIL message in their BFSDPOP algorithm is space-exponential in the number of cross-edges in the BFS tree. They use a method which limits the number of these cross-edges, making their algorithm more efficient than the original DPOP algorithm. They show this experimentally by comparing the runtime and maximal dimensions of UTIL messages of the two algorithms when applied to three different DCOP problems. BFSDPOP has a significantly smaller runtime than DPOP as well as slightly smaller UTIL messages. They remark however that additional research is needed in order to further reduce the message size.

ConcFB is one of the most promising options suggested in recent literature. Especially as the problems grow larger and denser, ConcFB reportedly performs 2-3 times better in performance measures such as CPU time and number of messages sent [3]. This despite the fact that it uses some form of synchronization in its search process. Peri and Meisels show that an inconsistent view of the environment might be worse than some extra idle-time in the processing due to the extra computational effort required by backtracking the search [19]. All of this makes ConcFB one of the most promising options when it comes to complete algorithms in real time applications.

Adopt was the first complete DCOP algorithm presented in the literature. Even though it shares many features with ConcFB, such as being asynchronous and fully decentralized, it comes out short in terms of performance compared to ConcFB [3]. This is mostly due to ConcFB being a more advanced version, utilizing parallel processing and early pruning to a larger extent.

4.2 Applicability

In the previous sections we have reviewed seven algorithms and compared them based on their space complexity and communication required between agents. Nevertheless, the performance of each algorithm is heavily dependent on the properties of the problem at hand; one strategy of problem solving may work better than others, depending on the properties. To help scientists pick an algorithm which is most suitable for solving their problem, we have constructed a guideline, presented in Figure 5. Here we have created a decision tree-like structure, based on the setting in which each treated algorithm performs well. The recommendations are based on the structural properties described in section 3 and findings reported in section 4.1. We note that the applicability also depends on the computational resources

available. This is since some of the algorithms are highly parallelizable, but also since the memory and communication complexity varies between the algorithms.

SynchBB is suitable for small problems or educational purposes due to its simplicity (see Figure 5). In problems where the agents cannot be ordered, OptAPO is recommended since it is the only algorithm that does not require a pseudo-tree structuring of the agents. Similarly, PC-DPOP is a good choice if partial sharing of information is at the nature of the problem. In general, ConcFB performs better on large problems than other existing algorithms. The only scenario where it is not a good choice is when the resources for communication between agents is small, since the communication complexity of ConcFB is exponential. In this case, either BFSDPOP or PC-DPOP is recommended, depending on the amount of memory resources available.

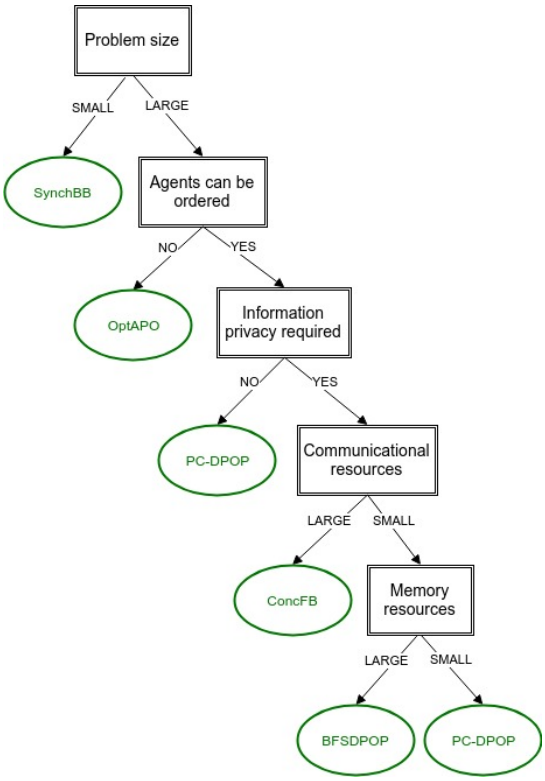


Fig. 5: Guideline for choosing a complete DCOP algorithm, depending on the properties of the problem and the available computational resources.

5 Conclusion

Seven algorithms have been reviewed and evaluated to clarify the current state of the field of complete DCOP algorithms. So far, the main efforts in improving the efficiency has been concentrated on increasing the amount of parallelization and early pruning of the search space.

The best performing complete DCOP algorithm to date is ConcFB, which can be ascribed to its heavily parallelized search structure and partial synchronization of knowledge between agents. The latter especially allows for restricting the search space by abandoning low potential solutions at an early stage. This feature increases the incentive to further investigate other options of partially synchronized algorithms.

However, it can be noted that most complete algorithms are currently not suitable for large scale problems due to their large computational needs. There is still a need to improve existing methods, possibly by further exploiting parallel processing within subproblems of the DCOP. Another idea might be to develop pre-processing methods that filter out poor assignment options from the domain of the variables in order to reduce the search space.

Even though ConcFB is the best performing algorithm in its own category of DCOP problems, we have seen that a wide range of algorithms can be considered the most suitable one depending on the structure of the problem. In conclusion, the problem needs to be carefully analysed before making a choice for a DCOP algorithm.

References

[1] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009.

[2] Victor Lesser, Milind Tambe, and Charles L. Ortiz, editors. *Distributed Sensor Networks: A Multiagent Perspective*. Kluwer Academic Publishers, Norwell, MA, USA, 2003.

[3] Arnon Netzer, Alon Grubshtein, and Amnon Meisels. Concurrent forward bounding for distributed constraint optimization problems. *Artificial Intelligence*, 193:186–216, 2012.

[4] Allan R Leite, Fabricio Enembreck, and Jean-Paul A Barthes. Distributed constraint optimization problems: Review and perspectives. *Expert Systems with Applications*, 41(11):5139–5157, 2014.

[5] Christopher Kiekintveld, Zhengyu Yin, Atul Kumar, and Milind Tambe. Asynchronous algorithms for approximate distributed constraint optimization with quality bounds. volume 1, pages 133–140, 08 2010.

[6] William Yeoh, Ariel Felner, and Sven Koenig. An asynchronous branch-and-bound dcop algorithm. *J. Artif. Intell. Res. (JAIR)*, 38:85–133, 05 2010.

[7] Ziyu Chen, Zhen He, and Chen He. An improved dpop algorithm based on breadth first search pseudo-tree for distributed constraint optimization. *Applied Intelligence*, 47(3):607–623, 2017.

[8] Zhen He and Zi-yu Chen. *BFSDPOP: DPOP Based on Breadth First Search Pseudo-Tree for Distributed Constraint Optimization*, pages 809–816. 2017.

[9] Katsutoshi Hirayama and Makoto Yokoo. Distributed partial constraint satisfaction problem. In *International Conference on Principles and Practice of Constraint Programming*, pages 222–236. Springer, 1997.

[10] R. Mailler and V. Lesser. Solving distributed constraint optimization problems using cooperative mediation. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, 2004. AAMAS 2004.*, pages 438–445, July 2004.

[11] Adrian Petcu and Boi Faltings. Dpop: A scalable method for multiagent constraint optimization. pages 266–271, 01 2005.

[12] Adrian Petcu and Boi Faltings. Pc-dpop: A partial centralization extension of dpop. 2006.

[13] Adrian Petcu. A class of algorithms for distributed constraint optimization. *Frontiers in Artificial Intelligence and Applications*, 194, 07 2009.

[14] Pragnesh Jay Modi, Wei-Min Shen, Milind Tambe, and Makoto Yokoo. Adopt: Asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1-2):149–180, 2005.

[15] William Yeoh, Ariel Felner, and Sven Koenig. Idb-adopt: A depth-first search dcop algorithm. In *International Workshop on Constraint Solving and Constraint Logic Programming*, pages 132–146. Springer, 2008.

[16] Toru Ishida, Les Gasser, and Hideyuki Nakashima. *Massively Multi-Agent Systems I: First International Workshop, MMAS 2004, Kyoto, Japan, December 10-11, 2004, Revised Selected and Invited Papers*, volume 3446. Springer, 2005.

[17] Ferdinando Fioretto, Enrico Pontelli, and William Yeoh. Distributed constraint optimization problems and applications: A survey. *CoRR*, abs/1602.06347, 2016.

[18] Alessandro Farinelli, Alex Rogers, Adrian Petcu, and Nicholas R Jennings. Decentralised coordination of low-power embedded devices using the max-sum algorithm. In *Proceedings of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 2*, pages 639–646. International Foundation for Autonomous Agents and Multiagent Systems, 2008.

[19] O Peri and A Meisels. Synchronizing for performance: Dcop algorithms. *ICAART 2013 - Proceedings of the 5th International Conference on Agents and Artificial Intelligence*, 1:5–14, 01 2013.

An overview of data science versioning practices and methods

Thom Carretero Seinhorst Kayleigh Boekhoudt

Abstract—Version control is important in software development, it makes it easy to manage collaboration between developers and to keep track of changes. Methods like Git and Subversion (SVN) are very popular, but are not ideal for controlling versions of data. This is mainly due to the size of the datasets used in data science and the amount of versions created. Data science therefore need its own versioning system so that data scientists can better collaborate, test, share and reuse data. There are numerous less familiar methods available for data science versioning, such as the Dataset Version Control System (DSVC), which is a system for multi-version dataset management, and the platform DataHub. In addition, there is also the method OrpheusDB, which is a dataset version management system that is built on top of relational databases. The storage representation is also an important aspect when it comes to data version control. In this paper we aim to compare the different methods for data science versioning. From the methods we compared, there is not one that is the ultimate best. Each method comes with its advantages and disadvantages, and can be used depending on the requirements of the research.

Index Terms—data science, data versioning, data control, DSVC, DataHub, OrpheusDB

1 Introduction

Version control systems have been around for a while and play an important role in software development [6]. It makes it easy to manage collaboration between developers and to keep track of changes. In a general workflow, all source code created in a software project is contained in a (de)centralized repository to which all project members have access. This version control system (VCS) allows for verifiability, quality and access control. Moreover, it allows for reverting unintentional changes. Version control in data science, however, is relatively new and has room for improvement [6]. Research in the field of data science can be a bit tricky and disorganised, especially because of the different stages involved such as collecting, exploring, cleaning and transforming the data, using the data to create a model and validating the model. These stages are often repeated multiple times. As a result of this cycle, you can end up with different versions of the data you started with. Data science therefore needs its own versioning system so data scientists can better collaborate, test, share and reuse data. These methods already exist and a couple will be discussed in this paper. Researchers want to collect, analyze and collaborate on datasets, to retrieve insights or to distil scientific knowledge from it [1].

A survey filled in by computational biology groups at MIT describes why a dataset versioning system is important [1]:

- Storing cost: teams of students, faculty and researchers share approximately 100TB of data via a networked file system which costs about $800/TB/year to store and maintain using a local provider for unlimited read/write access. This amounts to $100K/year
- Duplication: there are significant but unknown amounts of duplication of data. Simple file-level duplicate detection is insufficient since there is often some modification or extension in duplicates
- Deleting data: space (or cost) constraints lead to frequent request from the person in charge to reduce storage. This can cause stress since researcher do not know who is using their data or whether a particular dataset is essential for reproducibility of some experiment. Researchers would feel more comfortable deleting datasets if they knew that the dataset could be re-generated or if it were possible to track when a dataset had last been used.
- Retrieve versions: researchers would prefer a transparent mechanism to access data and write versions in order to be backward compatible with pre-existing scripts.
- Metadata management: researchers do not make heavy use of metadata management tools (like relational databases and wikis) to organize and share knowledge due to perceived cost of adding and maintenance of the data.

- *Thom Carretero Seinhorst is a MSC student at the University of Groningen.*
- *Kayleigh Boekhoudt is a MSC student at the University of Groningen.*

In this paper we aim to bring more awareness to data science versioning and compare different versioning methods. This paper is organized as follows. In section 2 we elaborate on related work in the field of versioning in data science. Followed by section 3 in which we discuss different data versioning methods. In section 4 we elaborate on different ways to store the data. Thereafter, we compare the methods discussed in this paper and make a conclusion about our research in section 5. Finally, future work will be discussed in section 6.

2 Related work

In software development, there are many well-known version control systems (VCS) available, for instance, Git and one of its platforms GitHub. Git and SVN (Subversion) have proved useful for collaborative source code management, but they are inadequate for managing datasets for several reasons. First, using a version control system, e.g. Git leads teams to resort to storing data in file systems, often using highly ad hoc and manual version management and sharing techniques. It is not uncommon to see directories containing thousands of files with names like data1-v1.csv, data1-v2.csv, data1-v1-after-applying-program-XYZ.txt, etc., possibly distributed and duplicated across multiple cloud storage platforms [1]. Second, the underlying algorithms are not optimized for large files or repositories, and can be painfully slow in such settings. Third, Git and SVN are based on a model of either ”checking out” the entire repository (Git), or keeping two copies of each file in the working directory (SVN). This may not be practical when dealing with large datasets. Fourth, Git and SVN employ Unix-diff-like differencing semantics when merging changes. For text files, this means they identify overlapping ranges of edits, and allow changes in non-overlapping regions. For relational datasets, this merge policy can be both too restrictive and miss conflicts [2].

Most data analytic software like SAS, Excel, R and Matlab lack dataset versioning management capabilities [1]. There are a few methods available for data science versioning, such as the Datahub MIT project and OrpheusDB, which we will discuss in this paper. There are also several startups and projects on providing basic dataset management infrastructure for data science applications, such as CKAN (ckan.org), Domo (domo.org), Enterprise Data Hub (cloudera.com/enterprise), Domino (dominoup.com), Amazon Zocalo and Dat Data (dat-data.com) [1]. To our knowledge, there has not been research that gives an overview and comparison of different methods that are available for versioning data.

3 DATA VERSIONING

A good practice to start with in data versioning is to treat data as immutable [6]. Save only one version of the raw data and never overwrite it. This is necessary for reproducibility. Researchers need to be able to use this data to replicate each others work. This is also important to restore processed data in the case of unintended actions or corruption.

After expensive and time consuming processing of data, one might want to export and save the results. Data can be difficult to version control due to its size and variety. The rise of the internet, smart phones and other technologies has produced a vast diversity of datasets from interactions on social media to medical records [1]. These datasets are diverse, varying from small to extremely large, from structured (tabular) to unstructured, and from complete to noisy and incomplete.

3.1 Ad hoc

Generated dataset versions are often stored in an ad hoc manner, typically in shared (networked) file systems, dumped to a disk or uploaded to a cloud [3]. A disadvantage of this practice is that this can easily use up the storage space on the disk or in the cloud. It is bad to waste storage space by saving different versions of slightly different data. For instance using 5GB to store data when you only changed 1 byte in the original 5GB file [6]. This challenge might not be that big of an issue when dealing with small amounts of data. However, in data science, data scientists repeatedly transform their datasets in many ways, by normalizing, cleaning, editing, deleting, and updating the data throughout the research. Another challenge of the ad hoc method is that it makes collaboration between researchers difficult. It takes a lot of effort to effectively manage and make sense of the data. The naming convention usually only makes sense to the person who created the files/chose the names, therefore it is difficult to determine the correct version. It is easy to make unintended changes, overwrite or delete data. This negatively affects the quality and reliability of the data. Finally, it is impossible to query across different versions of data using the ad hoc approach.

3.2 Database

In order to be able to query across datasets companies store their data in databases. One simple way of storing dataset versions would be to represent the dataset as a table in a database, and add an extra attribute corresponding to the version number [3]. However, this approach is extremely wasteful since each record is repeated as many times as the number of versions it belongs to. Data science teams often collaboratively analyze datasets, generating dataset versions at each stage of iterative exploration and analysis. There is a pressing need for a system that can support dataset versioning, enabling such teams to efficiently store, track, and query across dataset versions. While Git and SVN are highly effective at managing code, they are not capable of managing large unordered structured datasets efficiently, nor do they support analytic (SQL) queries on such datasets. However, source code version control systems are both inefficient at storing unordered structured datasets, and do not support advanced querying capabilities, e.g., querying for versions that satisfy some predicate, performing joins across versions, or computing some aggregate statistics across versions [1].

3.3 DSVC

Dataset Version Control System (DSVC) is a system for multi-version dataset management. DSVC's goal is to provide a common foundation to enable data scientists to capture their modifications and minimize storage costs. Further, DSVC makes it possible to use a declarative language to reason about versions, identify differences between versions and share datasets with other scientists. DataHub is a hosted platform, built on top of DSVC, that not only supports interaction capabilities, but also provides a number of tools for data cleaning, data search and integration, and data visualization [1].

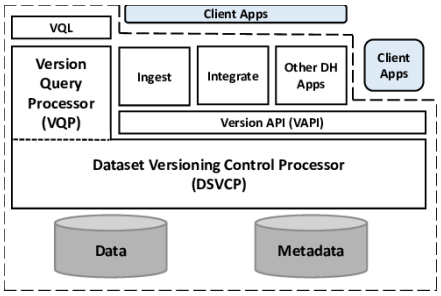


Fig. 1. Components and Architecture of DSVC and DataHub [1]

3.3.1 System Architecture

The high level architecture of DSVC is depicted in Figure 1. At the center of the DSVC there is the dataset version control processor (DSVCP), which processes and manages versions. The modeling of DSVC consist of two concepts:

- a table containing records
- a dataset consisting of a set of tables

For structured data, every record in for example a file is stored in a table. Each record is given a key and has a set of attributes associated with it, which is referred to as the schema. Each table has the same schema for every single record. For completely unstructured data, a single key is enough to refer to an entire file. The benefit of this model is that it offers flexibility by making it possible to store a wide range of data at different levels of structure.

A dataset consists of a set of tables, including relationships between them (e.g. foreign key constraints). The versioning information of datasets is stored in a version graph. A version graph is a directed acyclic graph where the nodes represent the datasets and the edges illustrate relationships between versions as well as provenance metadata provided. Provenance metadata is information concerning the creation of the version. It indicates the relationship between two versions and is generated by the user or automatically derived whenever a new version of a dataset is created. Examples of provenance metadata are [1]:

1. the name of the program that generated the new version
2. the commit id of the program in a code version control system like Git
3. the identifiers of any other datasets or data objects that may have been used in creating the new version

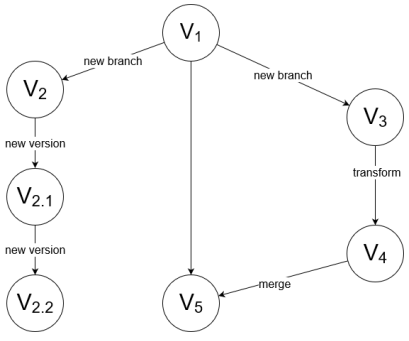


Fig. 2. Example of a versioning graph

A directed edge from node V_i to node V_j indicates that either V_j is a new version of V_i (e.g. $V_{2.1}$) or V_j is a new branch that is created as a copy of V_i and will evolve separately (e.g. V_3). The edge can also imply that V_j is obtained by applying an operation to V_i (e.g. V_4) or by the merging of branches (e.g. V_5).

3.3.2 Versioning API

DSVC provides a versioning API (VAPI) that, at a high level, is similar to Git API but comes with additional functionality. VAPI, similar to Git, contains the commands:

Git	
init	create a new local repository
checkout -b	create a new branch
merge	merge a different branch into your active branch
commit	commit changes to head (but not yet to the remote repository)
checkout –	undo local changes
clone	create a working copy of a local repository

DSVC	
create	create a new a dataset
branch	create a new version of a dataset
merge	merge two or more branches of the dataset
commit	make local (uncommitted) changes to the dataset permanent
rollback	undo local changes
checkout	create a local copy of a branch that is either a full copy, a lazily retrieved copy

Table 1. DSVC commands similar to Git

In addition to the commands above, the VAPI, like Git, allows users to specify hooks that fire off custom scripts when certain important actions occur. The VAPI hooks include checking commit messages for spelling errors, enforcing project coding standards, notifying team members of a new commit and pushing the code to production [4].

Furthermore, hooks notify applications to run off the newest version of a dataset (e.g. a dashboards plotting aggregated results), or tracking data products (results from statistical analysis) derived from the dataset. Hooks will then be used to re-run applications or update the data products. Instead of triggers, hooks can also be used to detect and correct errors.

3.3.3 Versioning query language

In addition to the API, DSVC supports a versioning query language named VQL, which is an enhanced version of SQL that allows users and applications to query multiple versions at once. VQL queries return results that are either data items from tables or pointers to versions (datasets).

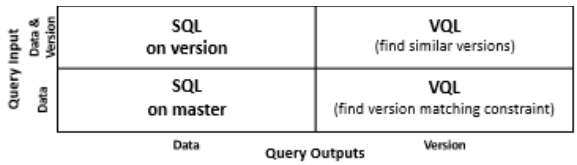


Fig. 3. DataHub query quadrant

Figure 3, shows four ways in which VQL can be used. The lower-left quadrant is the standard SQL query that (by default) will be executed on the master version. These types of queries have data (records) as input as well as output. It is also possible to execute queries for specific versions. This type of query falls in the upper-left quadrant. The query specifies data, but also one or more versions. On the other hand, it is also possible to retrieve version numbers. For instance, it is possible to return all the version numbers that meet a certain predicate based on data (bottom-right quadrant). Furthermore, it is possible for VQL to return version numbers that meet a predicate based on versions (top-right quadrant). You can, for example, ask DataHub to return the version numbers of a dataset that differ X amount of records with a specific version of the dataset.

3.4 OrpheusDB

While Git and SVN (Subversion) are great at source code version control, they are unfortunately unable to efficiently support large unordered datasets. Moreover, they cannot support the full range of operations supported natively by SQL.

OrpheusDB is a dataset version management system that is built on top of standard relational databases. It inherits much of the same benefits of relational databases, while also compactly storing, tracking, and recreating versions on demand [3]. OrpheusDB has been developed as open-source software (orpheus-db.github.io) and is an offshoot of the MIT DataHub project. OrpheusDB is a hosted system that supports relational dataset version management, with three design innovations. To start with, OrpheusDB is built on top of a traditional database. It is thus inherits all of the standard benefits of relational database systems. Secondly, OrpheusDB supports advanced querying and versioning capabilities, via both SQL queries and git-style version control commands. Lastly, OrpheusDB uses a sophisticated data model, coupled with partition optimization algorithms, to provide efficient version control performance over large-scale datasets.

3.4.1 System Architecture

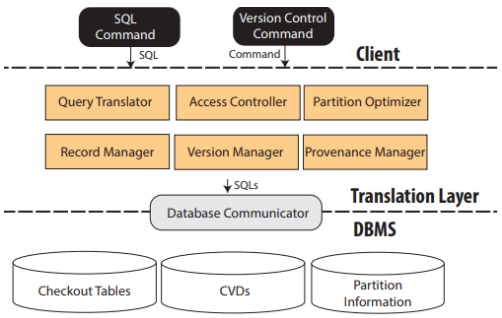


Fig. 4. Components and Architecture of OrpheusDB [3]

The high level architecture of OrpheusDB is depicted in Figure 4. At the center of OrpheusDB there is a collaborative versioned dataset (CVD) to which one or more users can contribute. Each CVD corresponds to a table [3], which is a set of tuples (records with attributes). The CVD essentially contains many versions of that table. A version is an instance of the table, specified by the user and containing a set of records. Versions within the CVD are related to each other in a similar way as the DSVC via a version graph. The version graph represents how the versions were derived from each other.

Records in a CVD are immutable, i.e., any modifications to any record attributes result in a new record, and are stored and treated separately within the CVD [3]. In general, there is a many-to-many relationship between records and versions. Each record can belong to many versions and each version can consist of many records. Each version has a unique version id (*vid*) and each record has its unique record id (*rid*), which is used to identify immutable records within the CVD and are not visible to end-users of OrpheusDB [3]. Furthermore, the table corresponding to the CVD may have primary key attributes that uniquely specify tuples. This implies that for any version no two records can have the same values for the primary key attribute(s). However, across versions, it is possible for multiple records to have the same the primary key attribute(s). OrpheusDB can support multiple CVDs at a time.

OrpheusDB consists of six core components [3]:

- the query translator: responsible for parsing input and translating it into SQL statement understandable by the underlying database system
- the access controller: monitors user permissions to various tables and files within OrpheusDB

- the partition optimizer: responsible for periodically reorganizing and optimizing the partitions via a partitioning algorithm LyreSplit along with a migration engine to migrate data from one partitioning scheme to another
- the record manager: records and retrieves information about records in CVDs
- the version manager: records and retrieves versioning information, including the *rid*s each version contains as well as the metadata for each version
- the provenance manager: responsible for the metadata of uncommitted tables or files, such as their parent version(s) and the creation time

The underlying Database Management System (DBMS) maintains CVDs as well as metadata about versions. In addition, the underlying DBMS contains a temporary staging area consisting of all of the materialized tables that users can directly manipulate via SQL without going through OrpheusDB [3].

3.4.2 OrpheusDB API

Users interact with OrpheusDB via the command line, using both git-style version control commands, as well as SQL queries [3]. To make modifications to versions, users can do two things. Users can either use SQL operations issued on the relational database that OrpheusDB is built on top of, or can alternatively operate on them using programming or scripting languages [7]. There are version control commands that users can use on CVDs much like they would with source code version control. OrpheusDB supports an important subset of Git commands enabling checkout, commit, init, create_user, config, whoami, ls, drop, and optimize. The *checkout* operation materializes a specific version of a CVD as a newly created regular table within a relational database that OrpheusDB is connected to. The *commit* operation adds a new version to the CBVD by making the local changes made by the user on their materialized table visible to others. In addition to checkout and commit, OrpheusDB also supports other commands:

- *diff*: a standard differencing operation that compares two versions and outputs the records in one but not the other
- *init*: initialize either an external csv file or a database table as a new CVD in OrpheusDB
- *create_user, config, whoami*: allows users to register, login, and view the current user name
- *ls, drop*: list all the CVDs or drop a particular CVD
- *optimize*: OrpheusDB can benefit from intelligent incremental partitioning schemes (enabling operations to process much less data)

In order to support data science workflows, OrpheusDB additionally supports the use of checkout and commit into and from csv (comma separated value) files via slightly different flags. The csv file can be processed in external tools and programming languages such as Python or R, not requiring that users perform the modifications and analysis using SQL.

OrpheusDB supports the use of SQL commands on CVDs via the command line using the *run* command, without having to materialize the appropriate versions. Users can run SQL commands on CVDs which either takes a SQL script as input or the SQL statement as a string. These SQL commands use the special keywords: VERSION, OF, and CVD. Moreover, users can use SQLs to explore versions that satisfy some property by applying aggregation grouped by version ids. When writing SQL queries, users can be entirely unaware of the exact representation, and instead refer to attributes as if they are all present in one large CVD table. Internally, OrpheusDB translates these queries to those that are appropriate for the underlying representation.

4 Storage Representation

Storage representation is also an important aspect when it comes to data version control. Both methods DSVC and OrpheusDB use versioning graphs to store relationships between versions and provenance metadata. However, the data can be stored using different representations. Bhardwaj, et al. [1] discuss two methods to store data using DSVC. OrpheusDB uses a hybrid approach for representing CVDs [3].

4.1 DSVC - Version-First Representation

For each version, the collection of records that are part of that version are stored in the storage graph. This is the most logical representation because it makes it easy for users to view all of the records in a particular version. This is one of the possible ways to store data of different versions using DSVC.

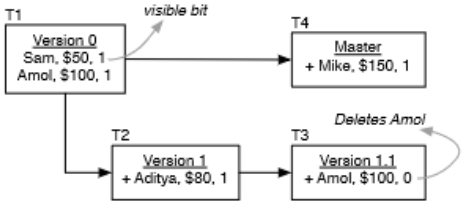


Fig. 5. Example of relational tables to encode 4 versions, with deleted bits [1]

In short, a branching history of versions is encoded with one or more materialized versions (retrieved from other versions) and a collection a deltas (differences) representing non-materialized versions. The retrieval queries can be answered by walking the storage graph appropriately. This method is relatively simple to implement. Whenever the user performs a branch command, a new table, which has the same schema as the base table, is created to represent the changes made to the new branch. In addition, each record is extended with a bit that allows the user to track whether the record is active in a particular version [1]. For instance, the extra bit equals 1 if a new record has been inserted and 0 if a record has been deleted. Figure 5 shows an example of this type of representation in a storage graph. This storage graph has two branches. The "Master" branch and the branch "Version 1". The original data (Version 0) is a table containing two records. At the head of the "Master" branch, the table consists of the names Sam, Amol and Mike (notice that Mike has been marked as added). At the head of the "Version 1" branch (labeled "Version 1.1"), the table contains the names Sam and Aditya (notice that Amol has been marked as deleted).

This approach is still a work in progress and has a few challenges [1]. First of all, finding the best way to store the delta between versions so that users can retrieve one version using the other and the delta. In figure 5, the extra bit is enough to retrieve the data in the tables because the tables were created using INSERT/DELETE commands. But, it is also possible to create new tables using more complicated scripts. The problem of efficiently encoding a graph of versions is also challenging. Just because two versions are adjacent in the versioning graph does not mean that they should be stored as differences against each other. If the difference between two non-adjacent versions (V_i and $V_{j.1}$) is smaller than two adjacent versions (V_j and $V_j.1$) than it might be more efficient to store $V_{j.1}$ against V_i instead of V_j.

4.2 DSVC - Record-First Representation

For this approach, data is encoded as a list of records, each annotated with the version it belongs to [1]. This is another way to store data using DSVC. The table below shows the record-first representation of the storage graph displayed in Figure 5. The advantage of this representation over the version-first representation is that it makes it easier to retrieve the versions with records that satisfy a certain properties. For example, it takes less effort to find the versions that contain records of people with daily wages higher or equal to $100 using this approach compared to the version-first method.

Name	Daily Wage	Version
Sam	50	{Version 0, Master, Version 1, Version 1,1}
Amol	100	{Version 0, Master, Version 1}
Mike	150	{Master}
Aditya	80	{Version 1, Version 1,1}

Table 2. Record first representation of the storage graph in Figure 5

Another example of a record-first representation is a temporal database, which is a database capable of storing data that is time-based. Medical applications, for example, may be able to benefit from temporal database support. A record of a patient's medical history can have little meaning unless the test results, e.g. the temperatures, are associated with the times at which they are valid. This might be necessary to help clarify why the patients' temperature changed at a certain period in time. On the other hand, this representation is not ideal for retrieving records of a specific version.

4.3 OrpheusDB - Split-by-rlist

rid	Protein1	Protein2	Neighborhood	Cooccurrence	Coexpression
r_1	ENSP273047	ENSP261890	0	53	0
r_2	ENSP273047	ENSP235932	0	87	0
r_3	ENSP300413	ENSP274242	426	0	164
r_4	ENSP309334	ENSP346022	0	227	975
r_5	ENSP273047	ENSP261890	0	53	83
r_6	ENSP332973	ENSP300134	0	0	83
r_7	ENSP472847	ENSP365773	225	0	73

vid	rlist
v_1	$\{r_1, r_2, r_3\}$
v_2	$\{r_2, r_3, r_4\}$
v_3	$\{r_3, r_5, r_6, r_7\}$
v_4	$\{r_2, r_3, r_4, r_5, r_6, r_7\}$

c.ii. Split-by-rlist

Fig. 6. Split-by-rlist for protein interaction data [5]

OrpheusDB uses a hybrid of split-by-rlist and a table-per-version to represent CVDs [3]. The split-by-rlist approach separates the data from the versioning information into two tables as shown in Figure 6. The first table, the data table, is a collection of all the records that appear in the versions. The second table, the versioning table, stores which version contains which record. The versioning table has the primary key attribute *vid* and the attribute *rlist*, which contains an array of the records present in that particular version. This approach allows easy insertion of new versions without having to modify existing version information. However, the checkout time grows with the size of the dataset [3] because it requires a join between the two tables to retrieve any version records. On the other hand, a-table-per-version approach, where each version is stored in a separate table, has very low checkout times because it only requires retrieving all the records in a specific table [3].

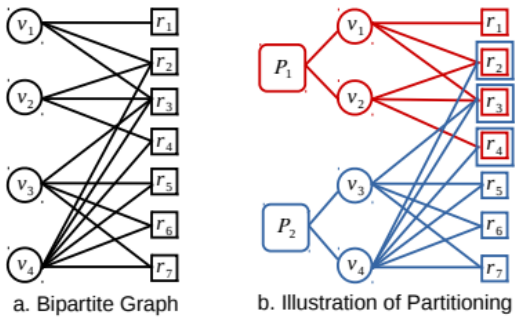


Fig. 7. Version-Record Bipartite Graph & Partitioning of Figure 6 [3]

Let $V = \{v_1, v_2, ..., v_n\}$ be the n versions and $R = \{r_1, r_2, ..., r_m\}$ be the m records in a CVD. The relationship between records and versions can be represented using a version-record bipartite graph $G = (V, R, E)$, where E is the set of edges. An edge between v_i and r_j exists if the version v_i contains the record r_j. Figure 7(a) illustrates the bipartite graph of the protein data used in Figure 6. The partitioning algorithm LyreSplit finds the best partitions $P = \{P_1, P_2, ..., P_s\}$ by minimizing checkout cost and storage cost [3]. The algorithm partitions G into smaller subgraphs denoted as $P_k = (V_k, R_k, E_k)$, where V_k, R_k, E_k represent the set of versions, records and edges in partition P_k respectively. Figure 7(b) shows a possible way to partition the bipartite graph in Figure 7(a). Partition P_1 contains version v_3 and v_4 while partition P_2 contains versions v_3 and v_4.

5 Conclusion

In this paper, we have explained four methods for the comparison of version control in data science: ad hoc, database, DSVC&DataHub and OrpheusDB. While Git and SVN are highly effective at managing code, they are not capable of managing large unordered structured datasets efficiently, nor do they support analytic (SQL) queries on such datasets. Table 3 gives an overview of the advantages and disadvantages of the data versioning methods discussed in this paper. From this we conclude that, when collaborating with others the best solution is either DSVC or OrpheusDB. If the data is already stored in a relational database then we recommend researchers to opt for OrpheusDB over DSVC because it is an add-on can be easily used with this type of database. No substantial changes is necessary. We only encourage using the ad hoc approach when the datasets are relatively small and you know that you will not be making a lot of changes to the data. In this case, we would also advise researchers to save the code that generated the datasets. If the ad hoc method is not suitable because it is also necessary to query over the datasets, we suggest researchers to use a relational database to store the versions in tables.

Method	Advantages	Disadvantages
Ad hoc	• quick solution • easy to store, for example in a file system or cloud	• difficult to collaborate with others • not ideal for large datasets (use up storage space) • difficult to keep track of changes • easy to make unintended changes (delete/overwrite)
Database	• query across datasets	• difficult to collaborate with others • a table per version is not ideal for large datasets
DSVC + DataHub	• minimize storage cost • query across different versions • able to keep track of changes • two possible ways to store data (version-first, record-first) • supports Git like commands and SQL like queries (VQL)	• designed to support data versioning "from the ground up" and not easily combined with existing relational database • finding the best way to store the delta between versions so that users can retrieve one version using the other and the delta • finding the best way to efficiently encode a graph of versions
OrpheusDB	• query across different versions • able to keep track of changes • supports collaborative data analytics using a traditional relational database • the storage representation combines minimizing the checkout cost and the storage cost • supports Git like commands, SQL and programming languages such as Python and R	• the foundation is an existing relational database

Table 3. Advantages and disadvantages of the different methods

We realize this is just the beginning of a new era of version control in data science and we expect many more of these methods to appear, however we do not expect the core principles to differ much from what we discussed.

6 Future work

For future work on data versioning systems, we recommend the following options:

- Research the best way to represent/store the delta using version-first approach (DSVC)
- Research the best storage representation for DSVC (record-first versus version-first
- Compare the storage representation of data versioning methods based on storage cost
- Compare the storage representation of data versioning methods based on checkout cost
- Research other data versioning methods that are available, for example for less structured data

References

[1] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. Parameswaran. Datahub: Collaborative data science & dataset version management at scale, September 2014.

[2] A. Chavan, S. Huang, A. Deshpande, A. J. Elmore, S. Madden, and A. Parameswaran. Towards a unified query language for provenance and versioning, June 2015.

[3] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. Parameswaran. Orpheusdb: Bolt-on versioning for relational databases, March 2017.

[4] M. Hudson. Git hooks, 2019.

[5] D. Szklarczyk, A. Franceschini, M. Kuhn, M. Simonovic, A. Roth, P. Mnguez, T. Doerks, M. Stark, J. Muller, P. Bork, L. J. Jensen, and C. von Mering less. The string database in 2011: functional interaction networks of proteins, globally integrated and scored., January 2011.

[6] S. Wang. Versioning data science, 2017.

[7] L. Xu, S. Huang, S. Hui, A. J. Elmore, and A. Parameswaran. Orpheusdb: A lightweight approach to relational dataset versioning, May 2017.

Selecting the optimal hyperparameter optimization method: a comparison of methods

Wesley Seubring, Derrick Timmerman

Abstract— One of the key-problems in the field of machine learning is the optimisation of hyperparameters. There exist several optimisation methods to automate this procedure. This paper compares the well-known hyperparameter optimisation methods Grid Search and Random Search to newer and more advanced methods such as Spectral Analysis, Bayesian Optimisation and CMA-ES Optimisation. The findings give insight in when the methods are best used in practice.
The insights are obtained by comparing efficiency, scalability and usability in practice for each optimisation method. Efficiency is measured by the speed of a method and scalability by its possibility to scale across multiple machines. For usability, we estimated how easily each method is put into use.
The results of this research show that all methods have their own uses within the field of machine learning. Although Grid Search and Random Search are useful methods, the remaining methods show improvement if the required expertise is available. Spectral Analysis is the most efficient method for parallel hyperparameter optimisation, while Bayesian Optimisation is most efficient for sequential hyperparameter optimisation. Grid Search is ideal for lower dimensional ($D < 2$) search spaces, and Random Search is an overall solid method that scales well. CMA-ES, however, is especially attractive for larger data sets, given sufficient time is available.

Index Terms— Hyperparameter optimisation, machine learning, model selection, Grid Search, Random Search, Spectral analysis, Bayesian Optimisation.

1 Introduction

During the last couple of years, machine learning models have shown an increase in complexity in order to deal with the explosively increased amount of available data. This complexity comes with an increased number of hyperparameters that have to be optimised for a given model. These so-called "hyperparameters" are the inputs for a machine learning algorithm that control for example the complexity of a model, but they can also be parameters that specify the learning algorithm itself. An example of a hyperparameter is the learning rate in Stochastic Gradient Descent that controls how large of a step to take in the direction of the negative gradient. However, as opposed to regular parameters, which are optimised by the algorithm itself, hyperparameters are fixed settings and initialised prior to the learning process of a model. Therefore, hyperparameters have to be determined in advance by a professional or more preferably by an algorithm.

The quality of a machine learning model, i.e. the ability of a model to correctly predict or classify unseen data, depends critically on the configuration of hyperparameters [9]. Let us take the learning rate in Stochastic Gradient Descent as an example again. Using a learning rate that is too high may overshoot the solution as it can cause the algorithm to miss a local optimum and diverge [3], while using a too low learning rate drops the speed of convergence drastically [5]. Therefore, it is important to obtain the optimal hyperparameters. However, as the number of possible combinations grows exponentially with the number of hyperparameters, selecting the optimal combination is not a straight-forward process [7]. This problem, the identification of good values for hyperparameters, is called the problem of hyperparameter optimisation [2].

The naive approach to determine the optimal hyperparameter values would be to manually try different values for each hyperparameter and apply these combinations to a validation data set, to estimate the best performing one. Fortunately, there are optimisation methods available that automate this procedure. With these methods, large amounts of different (i.e. unique) combinations can be evaluated in a relatively short amount of time compared to doing this manually. However, which method to use is not always a trivial decision as each method comes with its own pros and cons.

The aim of this paper is to serve as a guide in the field of machine learning, that could support professionals in their decision regarding an optimisation method.

While other algorithms are available, we only focus on five different methods: the two most commonly used ones Random Search and Grid Search [2, 10], Spectral Analysis [7], Bayesian Optimisation [8] and Co-variance Matrix Adaptation Evolution Strategy [18]. The first two methods are well-known and already often applied to machine learning models, whereas the last three methods are relatively new and are less frequently applied to machine learning models in practice [14].

These different methods are explored in depth and compared against each other, based on efficiency, scalability and usability in practice. By comparing these three characteristics we assume sufficient information is presented to give the professional a first impression of the best optimisation method to choose for a given problem. In addition to these three characteristics, we could also have included the comparison of the actual quality of the set of optimal hyperparameters. However, a simulation study would be necessary. Therefore, only efficiency, scalability and usability in practice are taken into account, which are measured as follows:

1. *Efficiency*: the efficiency of a method is based on how rapidly it determines the optimal hyperparameters. Scalability is not taken into account while estimating the efficiency.
2. *Scalability*: the scalability of a method is measured by its ability of parallel processing.
3. *Usability in practice*: the usability in practice of a method is based on how easy a method can be included or applied to a problem, e.g. the availability of methods in packages or frameworks.

In addition to the methods evaluated in this paper, more hyperparameter optimisation methods exist. However, these are not the focus of

- *Wesley Seubring, E-mail: w.seubring@student.rug.nl.*
- *Derrick Timmerman, E-mail: derricktimmerman92@gmail.com*

Manuscript received 9 April 2019.
For information on obtaining reprints of this article, please send e-mailto:derricktimmerman92@gmail.com.

the current work. Examples of these optimisation methods are:

1. *Gradient Based*: method for optimising multiple hyperparameters, based on the computation of the gradient of a model-selection criterion with respect to the hyperparameters. [15, 1]

2. *Population Based*: as stated in [11], Population Based Training (PBT) is a joint learning process that combines both hyperparameter search and model training into one single loop. Therefore, the outcome of PBT is not only a hyperparameter schedule but also a set of high performing models.

3. *Bandits*: Bandits is an optimisation method that focuses on speeding up Random Search through adaptive resource allocation and early-stopping. An example algorithm for this framework is HYPERBAND [9].

4. *Radial Basis Function Optimisation*: A deterministic and efficient hyperparameter optimisation method that employs radial basis functions as error surrogates. This optimisation method is optimised for deep learning algorithms as there many fewer function evaluations required compared to Bayesian Optimisation, which are incredibly computationaly expensive for a deep learning algorithm [12].

The paper is organised as follows: in section 2 the hyperparameter optimisation problem is explained with some additional information to ensure the reader is able to grasp the different concepts. In section 4 the optimisation methods are explained with some of their most important advantages and disadvantages. Then, in section 5 these methods are compared to each other and this paper ends with a conclusion in section 6.

2 Background

The aim of hyperparameter optimisation in machine learning is to find the hyperparameters of a given machine learning method that return the best performance as measured on a validation set. The hyperparameters need to be tuned by the engineer implementing the the machine learning method.

The hyperparameter optimisation problem can be formulated as:

$$\gamma^* = \underset{\gamma \in \Gamma}{\operatorname{argmin}} f(\gamma) \tag{1}$$

Where $f(\gamma)$ is an objective score function, such as root mean-square-error or the error rate on the validation set. γ^* is the set of hyperparameters that yield the lowest value of the objective function f which indicates the optimal configuration of hyperparameters for a given data set.

The problem with hyperparameter optimisation is that evaluating f can become extremely expensive, as each γ requires a certain amount of time to estimate its score. To compute this score, for each γ a machine learning model is first trained on a training set, and subsequently validated on a validation dataset, to compute the objective score (Figure 1). Eventually, γ is chosen that yields the lowest objective score. The time this process takes depends for a large extent to the size of a network that has to be trained. For example, a Neural Network with 20 hidden units requires less time than a Neural Network with 500 hidden units.

To account the difficulty of computing the objective function, methods often mention that they assume oracle mode, which describes that the objective function f is expensive to compute.

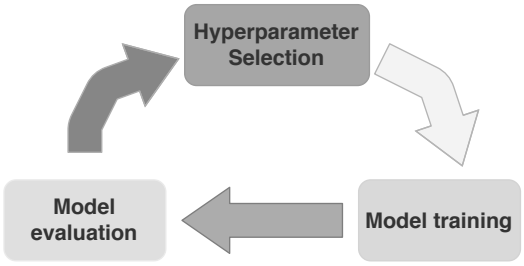


Fig. 1. Cycle of the Hyperparameter optimisation process for the training of a machine learning model.

3 Cross-validation

In [16] the relation between hyperparameter optimisation and cross-validation is described as follows:

The goal of hyperparameter optimisation is to come up with a model based on a set of hyperparameters that yields the best generalisation ability with respect to novel data. Given a set of hyperparameters, the model is determined by using training data set D. To achieve our goal, we have to find the best set of hyperparameters, and therefore a validation set V is necessary, that is independent of D. As the size of data is severely limited; we employ the procedure of cross-validation.

The procedure of cross-validation divides given data D at random into S distinct segments $\{G_s, s = 1, ..., S\}$, and uses S - 1 segments for training, and uses the remaining one for the test. This process is repeated S times by changing the remaining segment, and the generalisation performance can be evaluated by using for example the MSE (mean squared error) over all test results:

$$MSE_{CV} = \frac{1}{N} \sum_{s=1}^{S} \sum_{\mu \in G_s} (y^{\mu} - y(x^{\mu} | \hat{\theta}_s))^2 \tag{2}$$

Here G_s denotes the s-th segment for the test, and $\hat{\theta}_s$ denotes the optimal set of hyperparameters obtained by using D - G for training.

The extreme case of S = N (where N is the number of examples in the data set) is known as the leave-one-out (LOO) method, often used for a small size of data [17, 16].

By using cross-validation, the hyperparameters are optimised so that the cross-validation error MSE_{CV} is minimised.

4 Optimisation methods

Different methods exists for finding the optimal configuration of hyperparameters. The methods vary from each other in the way how they determine the optimal value for each hyperparameter.

4.1 Grid Search

From all hyperparameter optimisation methods, Grid Search is the most naive one in the way that it searches through the entire grid of possible hyperparameter combinations to eventually return the optimal one. It is basically the same as trying different combinations of hyperparameters manually and estimate their respective scores, but then in an automated way.

As an example, assume the training of a model for a Neural Network with 'number of layers' and 'number of hidden units' as the hyperparameters that need to be optimised. In order to perform Grid Search, it is necessary that fixed sequences of values are determined for both hyperparameters, in order to create and train models for each unique combination (see Table 1).

Each combination of hyperparameters correspond to a single model (see Table 2), that can be said to lie on a point of a 'grid' as illustrated

Hyperparameter	Values
Number of layers	[1,2,4]
Number of hidden units	[4,8,16]

Table 1. Examples of hyperparameters

	Number of layers	Number of hidden units
Model A	1	4
Model B	1	8
Model C	1	16
Model D	2	4
Model E	2	8
Model F	2	16
Model G	4	4
Model H	4	8
Model I	4	16

Table 2. Example of models which consist of unique combinations of hyperparameters.

on the left illustration in Figure 2. The goal is to train each of these models and evaluate their objective score. Based on the score of each model, the set of hyperparameters associated with the model that has the lowest objective score is returned and considered to be the optimal one (see also Equation 1).

One benefit of Grid Search is that every possible combination of hyperparameters is estimated. This ensures that the eventual set of hyperparameters returned by Grid Search, performed best among all possible combinations of hyperparameters, as no combination is left uncovered. In addition, this method is very intuitively and can be done with a relatively short implementation. The downside, however, is that Grid Search suffers from the *curse of dimensionality* as the number of joint values grow exponentially with the number of hyperparameters [2]. This is especially the case if there are many hyperparameters to tune.

A characteristic of Grid Search that could be both advantageous and disadvantageous is the absence of feedback from previous combinations [4]. Especially with respect to values for hyperparameters that perform worse, as this means that, although some hyperparameter values have already shown to perform badly, Grid Search continues to use these values in next combinations. However, this behaviour could also be desirable. In [4] the following two motivations are stated for this: first, it may not feel save to use methods which avoid doing an exhaustive parameter search by approximations or heuristics. The other reason is that the computational time required to find good parameters by grid-search is not much more than that by advanced methods if there are only a few hyperparameters to optimise. Furthermore, the results from Grid Search are most reliable for lower-dimensional search spaces (i.e. 1-D or 2-D) [2, 7].

4.2 Random Search

Random Search is similar to Grid Search in the way that the range for possible values for each hyperparameter is specified by a grid. The difference between the two methods lies in the fact that Random Search does not perform an exhaustive search over all possible combinations of values, but rather tries random values according to a distribution within the given ranges. This method is more efficient than Grid Search as there is a much better subspace coverage compared to the inefficient subspace coverage provided by Grid Search, which is explained in Figure 2 [2].

Although it is not guaranteed that all possible combinations are tried, Random Search is surprisingly time-efficient for identifying the optimal of the hyperparameter [2]. Besides, Random Search has all the practical advantages of Grid Search (e.g. ease of implementation) and only trades a small reduction in efficiency in low-dimensional search spaces for a large improvement in efficiency in high-dimensional search spaces. Whereas Grid Search tends to give better results for low-dimensional search spaces, Random Search performs better on higher-dimensional search spaces ($D \geq 3$) [2].

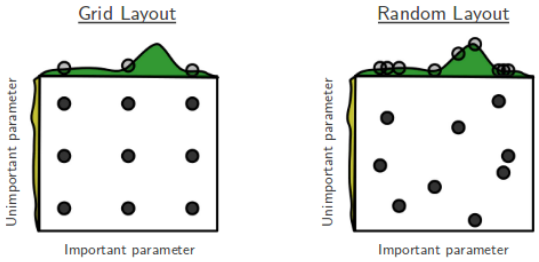


Fig. 2. This illustration shows how point grids and uniformly random point sets differ in how they cover subspaces. Projections onto either the $x1$ or $x2$ subspace, which is the case with Grid Search, produces an inefficient coverage as only three different parameter values from each of the parameters are tried, while with Random Search nine distinct values of each of the parameters are tried [2].

4.3 Spectral Analysis

Spectral Analysis is a newer and more advanced method used for hyperparameter optimisation. The method focuses on large sets of discrete parameters, however, such parameters are significantly more challenging that continuous hyperparameters [7]. To explain Spectral analysis, let:

$$f : \{-1,1\}^n \mapsto [0,1] \tag{3}$$

be a mapping from hyperparameter choices to test error. Where n is equal to the number of hyperparameters and {-1,1} represent the choice for each parameters as a binary number. Next is finding the optimal hyperparameters, such that f gives the lowest validation error. This is done, by making the assumption that each f can be approximated by a sparse low degree polynomial in the Fourier basis (i.e. an function with low dimensions and mostly zero values). Figure 3 illustrates how such an estimation build using Fourier series.

Using the assumption that f can be represented by this Fourier representation, Spectral analysis than applies sparse recovery for a set of uniform distributed samples for the hyperparameters configuration. With the sparse recovery an Fourier estimation of f is found. This new estimation of f will reduce the search space of the hyperparameters to an defined degree of complexity, using the largest coefficient's found for the samples. The estimation of f is used to find the optimal hyperparameters[7]. In practice this method is used in combination with Grid Search and Random Search which is done by narrowing the number of parameters by performing Spectral analysis for an limited number of stages. This reduces the search space to a workable size where after Random Search and Grid Search are applied to find the remaining hyperparameters.[7]

An advantage of Spectral Analysis is the significantly higher performance over Grid Search and Random Search, with speeds up to eight times faster than Random Search in higher-dimensional search spaces [7]. Disadvantages of Spectral Analysis are its availability (Table 3) and the need to optimise its own hyperparameters.

4.4 Bayesian Optimisation

Bayesian Optimisation is a powerful strategy for finding the optima of the objective function f [8]. The incorporation of a prior belief about the problem contributes for a large extent to the strength of this technique, as this shows higher probabilities of taking continuous steps

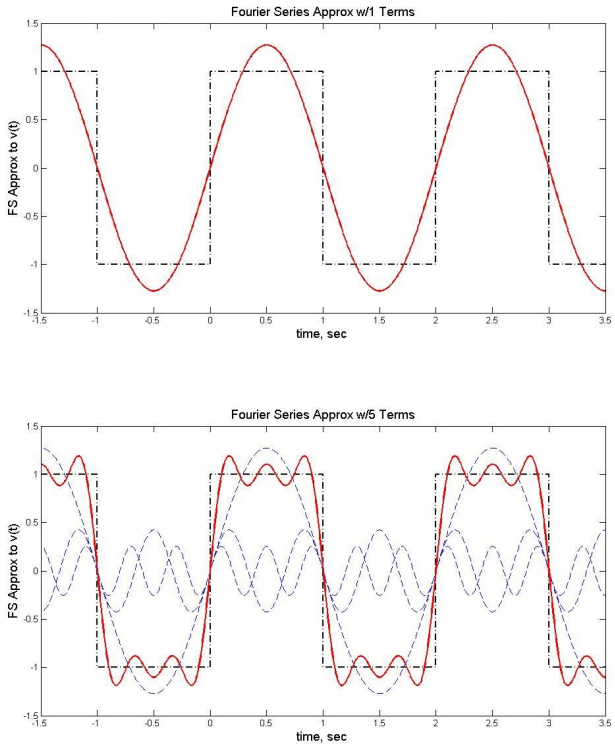


Fig. 3. This illustration shows estimation of a function using a Fourier series. Where the red line is the combination of all dashed blue lines.

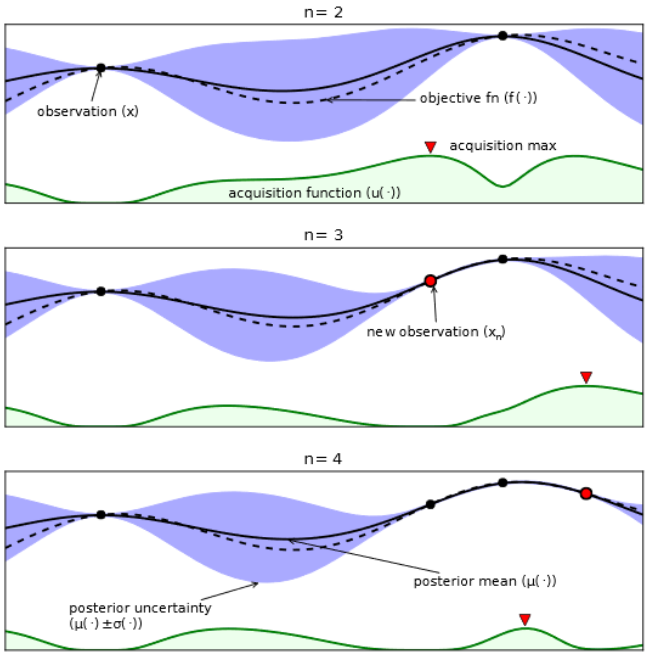


Fig. 4. Illustration of the Bayesian Optimisation procedure over three iterations. The plots show the mean and confidence intervals estimated with a probabilistic model of the objective function. Although the objective function is shown, in practice it is unknown. The acquisition function is high where the model predicts a high objective (exploitation) and where the prediction uncertainty is high (exploration). Note that the area on the far left remains unsampled, as while it has high uncertainty, it is correctly predicted to offer little improvement over the highest observation. [6]

in the right direction of the search space [8]. This method is called Bayesian because it uses "Bayes' theorem" which states that the *posterior* probability of a model θ given evidence E is proportional to the likelihood of E given θ multiplied by the *prior* probability of θ:

$$P(\theta|E) \propto P(E|\theta)P(\theta).$$

To make elaborate on this: results of combinations are used to construct a probabilistic model which maps hyperparameters to a probability of a score on the objective function:

$$P(score \mid hyperparameters)$$

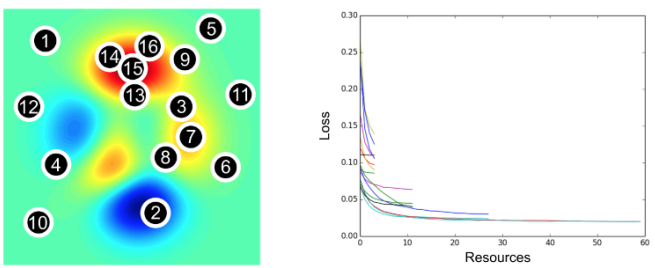
which is also referred to as a 'surrogate' for the objective function and is represented as:

$$P(y \mid x).$$

As already mentioned, Bayesian Optimisation has a prior belief about the objective function and iteratively updates this prior based on the results of previous combinations to eventually get a posterior that better approximates the objective function. As a surrogate is formulated, this becomes easier as it is easier to optimise the surrogate than it is for the objective function [8]. What Bayesian Optimisation does is, within each iteration, each next combination of values to be evaluated by the objective function, is based on the combination of values that performs best on the surrogate function.

So, instead of trying as many combinations as possible in order to find the optimal solution, the aim of this method is to become more accurate as the number of tried combinations increases. This goal is chased by updating the surrogate probability model within each iteration (i.e. after evaluation of the objective function) which is illustrated in Figure 5.

The advantage of this method is that each new combination is based on results of previous combinations. Using better-informed methods to choose the next hyperparameters, leads to less time spend on evaluating poor hyperparameter choices. However, this also means that this method has to be performed sequentially which rules out the use of parallelism.



(a) Configuration Selection (b) Configuration Evaluation

Fig. 5. (a) The heatmap shows the validation error over a two-dimensional search space with red corresponding to areas with lower validation error. The Bayesian method adaptively chooses new configurations to train, proceeding in a sequential manner as indicated by the numbers. (b) The plot shows the validation error as a function of the resources allocated to each configuration (i.e. each line in the plot). Configuration evaluation methods allocate more resources to promising configurations [9].

4.5 Co-variance Matrix Adaptation Evolution Strategy

Co-variance Matrix Adaptation Evolution Strategy (CMA-ES) can be used as an evolution based approach for the tuning of hyperparameter. CMA-ES performs best for larger function evaluation budgets [11]. In short, CMA-ES is an iterative algorithm that creates a set of candidates during each iteration from a multivariate normal distribution, which is subsequently validated. The validation results are then used to adjust the sampling distribution to a search space with a higher probability of providing better results.

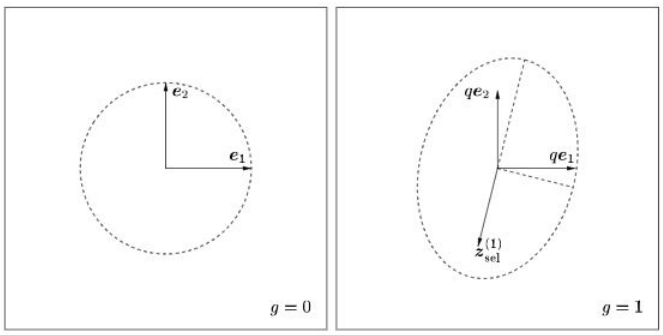


Fig. 6. Construction of the mutation distribution for CMA-ES for 2 hyperparameters. The first image shows the initial distribution of the mutation between the two hyperparameters. The second image shows how the mutation distribution is changed based on the best chosen hyperparameters z_{sel} and q is the decay weight for the previous evolution steps

Going in greater dept, CMA-ES builds an mutation distribution over time as a function of the most optimal results for now. Figure 6 shows the first two iterations of the mutation distribution. In the first generation $g = 0$ an equal mutation distribution between two hyperparameters is observed. Therefore, in the next generation both hyperparameters are equally likely to change as a form of mutation. With this mutation distribution and an initial set of hyperparameters, the first generation is created and validated. The best result of this generation Z_{sel} is added to the mutation distribution where after the previous results are multiplied by a decay factor q. In the next generation, the current best result is used and the new mutation distribution is applied. This process repeats itself for each generation. Figure 7 illustrates how the population changes over the generations.

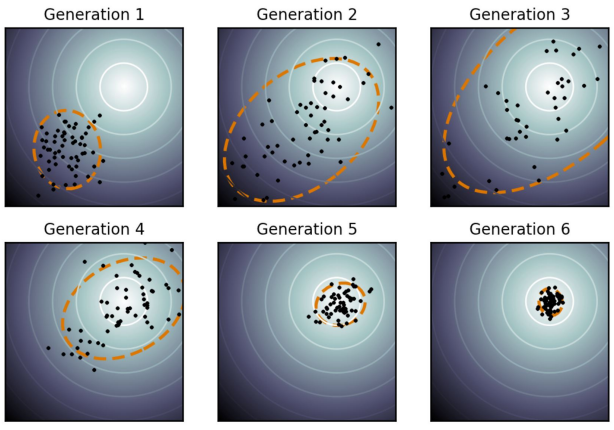


Fig. 7. Representation of the population over six generations. The dotted line shows the co-variance distribution used to create the population. You can see how the population concentrates over the optimum [18].

Advantages of this method are its robustness, the fact that it is computational cheap and that it supports parallel evaluation [11]. When running this algorithm on larger data sets for extensive period of time (e.g. the experiment done by I. Loshchilov et al. ran for 30.000 GPU days [11]), it will give outstanding performance compared to Bayesian Optimisation and Random Search. Since the computation of a new population is based on the co-variance matrix that is derived from the mutation distribution, a new population can be computed fast [18].

One of the limitations for this method is its usability for computation on a lower time budget, as CMA-ES performs worse with a shorter period of time [11]. In addition, for a smaller number of hyperparameters the results will not be great, due to the random mutation nature of the method.

5 Comparison

The different methods all have their own strengths and weaknesses, in this section we will compare them on efficiency and usability in practice.

5.1 Efficiency

To compare the efficiency between the optimisation methods, the speed of each method is taken into account. The speed of a method is measured by the number of tries it has to perform until the set of optimal hyperparameters is determined. For the comparison of efficiency, the use of parallelism is not incorporated.

The most naive method, Grid Search, is less efficient compared to other optimisation methods since it has to train and validate a model for each unique combination of hyperparameters in order to obtain the set of optimal hyperparameters. Therefore, it suffers greatly from the curse of dimensionality and is best used in lower dimension search spaces [2].

Random Search is more efficient in higher dimension search spaces, since the functions of interest have low effective dimensionality (e.g. a small subset of features determine the majority of the performance) [2]. Figure 2 shows how Random Search uses this to explore more of the subspace. This method is already more efficient compared to Grid Search, but it still requires long computational time for higher dimensions of hyperparameters. [2].

Spectral analysis reduces the search space by processing batches and using the properties of the Fourier basis, this allows to method to find optimal hyperparameters effectively, by searching over a compressed representation of the hyperparameters. Therefore, this method can become up to eight times faster than Random Search. However, for a smaller set of hyperparameters, Random Search and Grid Search can still out perform Spectral Analysis [7].

Bayesian Optimisation uses an objective function to find steps in the hyperparameter space that only improves the performance. For this to be used efficiently, human expertise is needed but with an equal amount of computation time Bayesian Optimisation is more likely to find good hyperparameters [8].

The last method uses evolution to increasingly find better performing hyperparameters. However, the random nature of this evolution makes the method less suitable for a smaller number of hyperparameter. As explained in subsection 4.5, outstanding performances are reached on larger data sets with extensive periods of time available for the method to perform. Therefore, this method is not an attractive method if time is limited as this method is not most efficient.

5.2 Scalability

In addition to the efficiency of a method, scalability also plays an important role in determining which method is best suitable for a given hyperparameter optimisation problem.

As explained in the previous section, Grid Search, is less efficient compared to the remaining optimisation methods. However, it is extremely well-suited for parallelism as objective scores for combinations of hyperparameter can be determined independently of each other. If the computational resources are available, Grid Search can make optimal use of these facilities.

Next to Grid Search, Random Search and Spectral Analysis are also parallel executable, such that the optimisation tasks can be preformed in different batches over parallel machines/cores. This is especially useful for the higher dimension search spaces, where computations could take a long period of time to perform on a single machine. While Grid Search, Random Search, Spectral Analysis and CMA-ES are all scaleable over multiple machines, there are differences. An advantage of Random Search over the remaining methods is that, when executed on multiple machine, it can cope with failing runs and change the size of the search grid on the fly [2]. This is due to the random nature of Random Search which makes it possible to start new nodes on the fly without many constraints. For Spectral Analysis, Grid Search and CMA-ES this is not possible.

The Bayesian Optimisation, however, is a sequential method that can not run parallel which makes it less suitable for extremely high dimensions. In such situation the optimisation is preferably executed in parallel to speed up the process [8].

5.3 Usability in practice

Usability plays an important role as it is necessary to support the application of hyperparameter optimisation and because an optimisation method that is considered to be efficient, does not necessarily have to be easily applicable in practice.

Grid Search is straightforward in its way of searching for the optimal combination of hyperparameters, as it naively checks all possible configuration. This method is mostly used in scientific research, due to its simplicity, reproducibility [2] and due to the fact that Grid Search is well-represented in a scientific language as Python (see Table 3). The method is an intuitive and understandable method, which allows users to implement Grid Search relatively easy.

Random Search shows similarities to Grid Search in the way that combinations of hyperparameters are selected from a grid. Besides, as explained in Section 4.1, Random Search shows to be very efficient through which it is considered by many professionals as the go-to method for tuning hyperparameters [10]. This is supported by the many different packages that include Random Search (see Table 3).

In contrast to the popularity of Grid Search and Random Search, Spectral Analysis is a less prominent player in the field of optimisation methods. This is not only observed by the limited amount of available literature among this method, but it is also shown in the lack of packages that include Spectral Analysis (see Table 3). One reason for this method to be under represented is the fact that this method is quite new. However, the biggest reason professionals are less likely to select Spectral Analysis as their optimisation method, has to do with the fact that Spectral Analysis itself also needs to be optimised. The implementation of Spectral Analysis actually requires the configuration of six hyperparameters that need to be tuned for getting the best performance. This could constitute a barrier to professionals as it initially takes more time to setup this method compared to e.g. Grid or Random Search.

Bayesian Optimisation is a more popular method compared to Spectral Analysis. Despite the fact that it includes some administrative overhead, as well as it requires expertise to get reasonable results [13], many professionals implement this method as it is efficient. Due to the optimisation of the surrogate during each iteration, the total number of iterations required is significantly smaller than one would need with e.g. Grid Search to reach the optimal solution. In addition, many papers exist that have done research on Bayesian Optimisation and combined with the presence of numerous available packages that include the Bayesian Optimisation (see Table 3), it is a widely appreciated approach by many professionals.

Method	Package name	Language
Grid Search	Scikit-learn[1]	Python
Grid Search	Talos [2]	Python
Grid Search	NMOF [3]	R
Random Search	Hyperas [4]	Python
Random Search	Hyperopt [5]	Python
Random Search	Hyperopt-sklearn [6]	Python
Random Search	Scikit-learn	Python
Random Search	Talos	Python
Spectral Analysis	Harmonica [7]	Python
Bayesian Optimisation	BOCS [8]	Matlab
Bayesian Optimisation	Auto-sklearn [9]	Python
Bayesian Optimisation	Scikit-optimize [10]	Python
Bayesian Optimisation	SMAC [11]	Python/Java
Bayesian Optimisation	MlrMBO [12]	R
CMA-ES	CMAES[13]	R
CMA-ES	CMA-ES/pycma[14]	Python

Table 3. Availability of methods

CMA-ES is an less prominent player in the field of hyperparameter optimisation methods. The method is available in a few standalone packages and is quite an advanced method to implement, due to the fact that its qualities are found in large scale problems. Parallelism need is not provided out of the box.

6 Conclusion

Different optimisation methods exist which each have their own strengths and weaknesses. This paper limited itself to the comparison of four different methods and discussed their efficiency, possible scalability and usability in practice.

Spectral analysis is the best method when we only look at efficiency, it has the best performance and scales well. Both are helpful in the higher dimensions of hyperparameters. For sequential computation Bayesian Optimisation is a good alternative.

When only considering the ease of use in practice, we see other methods giving better results. Spectral analysis is the least available method and has the most complex parameters of all the methods. We see that Grid Search and Random Search are the most well-known methods, widely available and simple in use. Bayesian Optimisation is also widely available, however but requires domain expert for implementation.

[1] https://scikit-learn.org/stable/whats_new.html#version-0-20
[2] https://github.com/autonomio/talos
[3] https://cran.r-project.org/web/packages/NMOF/index.html
[4] https://github.com/maxpumperla/hyperas
[5] https://github.com/hyperopt/hyperopt
[6] https://github.com/hyperopt/hyperopt-sklearn
[7] https://github.com/callowbird/Harmonica
[8] https://github.com/baptistar/BOCS
[9] https://github.com/automl/auto-sklearn
[10] https://github.com/scikit-optimize/scikit-optimize
[11] https://github.com/automl/SMAC3
[12] https://github.com/mlr-org/mlrMBO
[13] https://www.rdocumentation.org/packages/cmaes/versions/1.0-11
[14] https://github.com/CMA-ES/pycma

In conclusion we argue that there is no single best hyperparameter optimisation method for all types of optimisation problems. The methods we discussed all have their own advantages and disadvantages. These methods should be used accordingly, and more advanced methods should be used more in practice. Grid Search is recommended for the lower dimensional hyperparameters or if there are no limitations with respect to computational resources.

If the search for optimal hyperparameters should be done with the least effort and highest flexibility, Random Search is a good alternative due to its scalability and its general applicability. This method provides an overall good estimation of hyperparameters, is widely available and has descent performance.

As opposed to Grid Search, Spectral analysis is recommended when dealing with higher dimensions of hyperparameters and the amount of time available is sufficient to setup the method (e.g. implementation and hyperparameter optimisation). During our research, Spectral Analysis showed to be the most efficient method which could give an performance improvement up to eight times over Random Search, given the optimal circumstances.

CMA-ES would be a good solution, given the time available is sufficient as this method performs best on large scale problems while optimising for a longer period of time.

In addition to CMA-ES, if there is no or small time pressure, Bayesian Optimisation would also be a robust solution, as this is the best option for sequential computation, given the required profession is available.

These findings should give professionals a decent overview for the discussed hyperparameter optimisation methods. It should be clear when it is useful to use the more advanced methods and when withhold and go for the simpler methods.

While there is literature available for each optimisation method individually, only few research papers are available that compare several optimisation methods. With this paper we aim to fill this gap in our knowledge and provide a concise explanation and comparison of different well-known optimization methods. We compared the hyperparameter optimisation methods based on efficiency, scalability and availability, and therefore should give a general idea of the methods. In the section Future work, we will suggest a couple of extensions that could increase the quality of the comparison between the hyperparameter optimisation methods. Additionally, we provide an overview with existing packages for the different hyperparameter optimisation methods.

7 Future work

For future work we would suggest extending our research by adding more hyperparameter optimisation methods to our comparison such as Gradient Based Optimisation, Population Based Optimisation and Radial Basis Function Optimisation. Furthermore, an actual comparison in the form of a simulation study between the different methods, would be a valuable addition to our paper. With this addition, each method can be executed in an identical environment over the same data set. This would give us the opportunity to, besides estimating the speed of each optimisation method, also compare the actual quality of each set of optimal hyperparameter, provided by a method. For example, a highly-efficient optimisation method might provide a set of optimal hyperparameters which perform worse on a validation set compared to a more time consuming optimisation method.

Acknowledgements

The authors wish to thank Frank Blaauw et al. for reviewing this paper.

References

[1] Y. Bengio. Gradient-based optimization of hyperparameters. *Neural Computation*, 12(8):1889–1900, 2000.

[2] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *Journal of Machine Learning Research 13*, pages 281–305, feb 2012.

[3] L. B. M. Caio B. Nbrega. Predicting the learning rate of gradient descentfor accelerating matrix factorization. *Journal of Information and Data Management*, 5(1):94–103, Februay 2014.

[4] C.-C. C. Chih-Wei Hsu and C.-J. Lin. A practical guide to support vector classification. May 2016.

[5] C. Darken and J. Moody. Note on learning rate schedules for stochastic optimization.

[6] V. M. C. E. Brochu and N. de Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active usermodeling and hierarchical reinforcement learning. 2009.

[7] A. K. Elad Hazan and Y. Yuan. Hyperparameter optimization: A spectral approach. 2017.

[8] V. M. C. Eric Brochu and N. de Freitas. A tutorial on bayesian optimization ofexpensive cost functions, with application toactive user modeling and hierarchicalreinforcement learning. December 2010.

[9] L. L. et al. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research 18*, pages 1–52, April 2018.

[10] B. R. Horia Mania, Aurelia Guy. Simple random search provides a competitive approach to reinforcement learning. march 2018.

[11] I. L. . F. Hutter. Cma-es for hyperparameter optimization of deep neural networks. 2016.

[12] T. A. I. Ilievski, J. Feng and C. A. Shoemaker. Efficient hyperparameter optimization of deep learning algorithms using deterministic rbf surrogates. 2017.

[13] M. M. Ian Dewancker and S. Clark. Bayesian optimization primer.

[14] D. Y. J. Bergstra and D. D. Cox. Making a science of model search: Hyperparameter optimizationin hundreds of dimensions for vision architectures. 28, 2013.

[15] A. Jameson. Gradient based optimization methods. *MAE Technical Report No. 2057*, 2003.

[16] R. N. Kentaro Ito. Optimizing support vector regression hyperparameters based on cross-validation. *International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[17] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence (IJCAI)*, 1995.

[18] A. O. N. Hansen. Completely derandomized self-adaptation in evolution strategies. *Evolutionary Computation*, pages 159–195, 2001.

Reproducibility in Scientific Workflows: An Overview

Konstantina Gkikopouli and Ruben Kip

Abstract—Provenance, data of the chronology of steps in the workflow, is identified as an increasingly significant component in the life cycle of a scientific workflow, as it obtains in details all the occurred events of the system. Based on this gathered data, different actions can be performed (e.g. an execution history of a process or repair of the scientific workflow) that permit the reproducibility of different scientific applications. However, the capacity in terms of storage and computing needed to record all this required data in order to make the workflow reproducible still remains an unsolved issue. We summarize the current usage of Scientific workflow and its problems by creating an overview of recent papers, however we refrain from giving any general solutions. We mainly focus on scientific workflow integration in cloud computing. This integration assures the re-execution of the scientific workflows and their reproducibility in the cloud.

Index Terms—Scientific workflows, reproducibility, provenance, cloud computing.

1 Introduction

A workflow is a set of tasks and dependencies that are processed in a well-defined order to accomplish a specific goal[9]. In the context of science, workflows are used to describe scientific experiments. A scientific workflow is a flow of mostly computational tasks, that represent a significant part of a scientific experiment[9]. These workflows are highly applied in many fields including simulation, data analysis, image processing, and many other functions[9]. A basic scientific workflow example of processing a batch can be seen in figure 1.

Scientific workflow contains some unique characteristics[9], which make it special compared to the other types, e.g. business workflow. The scientific workflows tend to change regularly. As the scientists try to prove different hypothesis, they perform experiments, which might accidentally contain errors. Once they detect these faults, the scientists have to modify the corresponding steps, while the experiment proceeds. In contrast, the business workflows continue to be executed based on the same techniques and predefined standard methods. While being deployed and executed on distributed systems, the scientific workflows have to be adaptable to the system's dynamic nature, where the available resources may join and leave at any time. They are normally designed and constructed by scientists, who are experts in their corresponding fields and not necessarily specialists in the area of Information Technology. Furthermore, the scientific workflows may obtain complex control flow and data flow, which require a high-level language and an easy to use scientific workflow composition tool, in order to mitigate this complexity.

Two highly crucial components of scientific workflows is provenance and reproducibility(repeatability). Provenance contains valuable information regarding the origin, the history and all important events in the life cycle of a process. Based on these gathered data, the purpose of reproducibility is to reproduce(repeat) a process(or a part of an experiment) exactly the same way it was executed in the past, so as to obtain the same results. Reproducibility has a high significance regarding the scientific workflows. By reproducing an experiment, the scientists and the users are offered the opportunity to prove, reuse and verify the correctness of the experiment's results. In addition, the scientists are able to share their experimental methods and tools among each other, in order to have a better understanding and overview of their experiment. Another scenario would be that the scientists share their methods or results and re-use them as a building block for a new potential experiment. However, the capability of recording all the required data in order to make the workflow completely reproducible is an extremely hard problem. One of the most difficult challenges in making the scientific workflows fully reproducible is the heterogeneity of its components (components like soft/hardware etc.), which might often require different and conflicting sets of dependencies. Achieving a completely successful reproducibility of these workflows does not only depend on just sharing their specifications. It also relies on the capability of isolating necessary and sufficient computational artifacts, preserving them together with adequate description, so they can be again reused in the future [9].

Scientific workflows vary in size from a small number of tasks to millions of tasks. For large workflows, it is highly preferred to distribute this huge amount of tasks to different computers. The main reason for this is that the scientists and the users of the workflow want to complete the execution and to obtain their results as quickly as possible. As the scientific workflows require a huge computation power, cloud computing represents a new method for deploying and executing these workflows, as the cloud offers theoretically unlimited resources. On the one hand, clouds can be considered as another platform for carrying out workflow-based applications. They support the same methods for workflow management and execution that have been developed for clusters and grids. On the other hand, clouds also present many features, such as virtualization, that provide new capabilities for an easier deployment, management and execution of the workflow applications.

In this paper we consider the impact of cloud computing on scientific workflow applications, emphasizing on their reproducibility. We consider cloud as it offers important advantages, like limitless resources, to scientific workflows. Furthermore we emphasize on reproduciblity, because as scientific workflows and relevant technologies are becoming more complex, the challenge of maintaining reproduciblity becomes also harder[3]. Firstly, a background is provided, where attempts of previous researchers and briefly discussed. After that,the main advantages and drawbacks of the clouds regarding the scientific workflows are analyzed and explained in section 5. Following that we discuss the requirements and importance of reproducibility in cloud workflows section **??**. Then some example deployment methods are given and illustrated in section 6. Lastly, sections 7 and 8 summarizes our work by mentioning the main results and we provide some potential researches regarding the scientific workflows in the cloud that might occur in the future.

2 Methodology

The aim of this paper is to mainly address the long-standing issue of reproducibility in scientific workflows. Based on the available online resources and literature presented in our references, our study focuses on making the readers familiar with the topic by providing firstly some basic concepts and by describing the current situation regarding the workflows. As cloud computing represents a sophisticated method of deploying and executing the scientific workflows, the paper links re-

- *Konstantina Gkikopouli, studying MSc Computing Science, E-mail: k.gkikopouli@student.rug.nl.*
- *Ruben Kip, studying MSc Computing Science, E-mail: g.r.kip@student.rug.nl.*

For information on obtaining reprints of this article, please send e-mail to one of the authors.

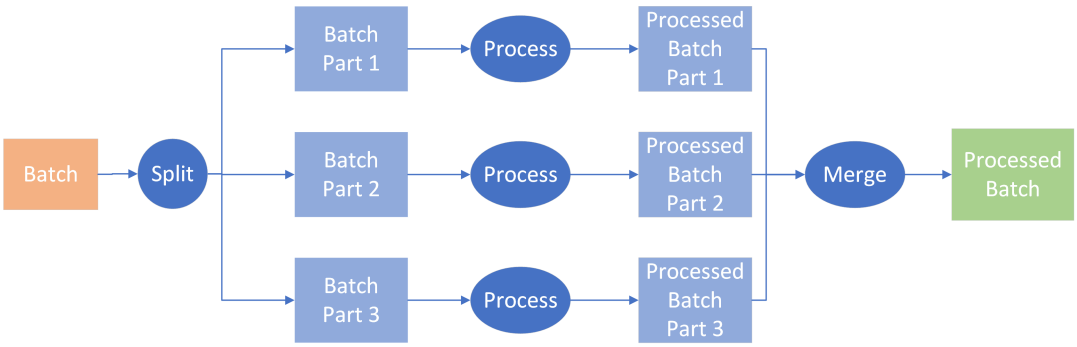


Fig. 1. Basic scientific workflow model

producibility with the cloud platform. We are mainly focused on the Infrastructure as a Service (IaaS) clouds, as they have been primarily used by the scientific workflows. Even though Platform as a Service(PaaS) clouds and Software as a Service (SaaS) clouds offer more advanced and sophisticated benefits to workflow-based computations, they lack of systems developed in this area and further research needs to be performed. The approach of the research is to offer a detailed overview of how the cloud can mitigate some of the issues of reproducibility in scientific workflows. In order to achieve our goal, we chose a number of resources, which served best our intention.

3 Background and Related Work

There has been a high amount of conducted researches, which focused on identifying the challenges in achieving workflow reproducibility. Shortly, the issues can be summarized as: insufficient and non-portable description of a workflow including missing details of the processing tools and execution environment, unavailable execution environments, missing third party resources and data, and reliance on external dependencies, such as external web services, which add difficulty to reproducibility at a later time[10].

Most of these conducted researches have emphasized on two different types of preservations. The first one is *physical preservation*, where the workflow is maintained by packaging its corresponding elements. As a result, this indicates the creation of a replica, which can be re-used later. The other one is *logical preservation*, where the workflow and its components have a detailed description, that can be reused by people, who want to reproduce a quite similar workflow[10].

There have been numerous attempts by different researchers and scientists, who tried to provide a potential solution to repeatability and reproducibility. One of them was Chirigati et al., who suggested *ReproZip* for the implementation packaging of workflows. The aim of this implementation is to package together the system calls during the workflow execution together with the dependencies, data and configurations involved during runtime. After that, the package can be used for the re-execution the registered workflow invocation.

Another method used for packaging the workflows was *Virtual machine as an image* (VMI). The main advantages of this approach is that the entire environment can be easily captured, shared with other scientists and re-executed. However, the eventual images are large in size, quite costly to be publicly distributed and do not capture a detailed and structured description of the entire computation [10].

When it comes to logical preservation, different researches were conducted. As it is before-mentioned, the logical preservation emphasizes on capturing all the details of the workflow and its components, which can later be used to reproduce a similar workflow. One example of this type of preservation is *myExperiment*[10]. It presents a web interface, which enables the sharing of computational description and visualization of the workflow and its corresponding elements. This approach has a positive impact on developing the reproducibility in the scientific workflows.

Also, an additional technique was provided by Santana-Perez et al., who proposed a semantic-based method to maintain scientific workflows, while they are being executed. In this case, the use of semantic libraries is really essential, as they identify all the resources that participate in the execution phase of a workflow. However, other researches have proved that this approach can not achieve a completely successful reproducibility of the workflows.

Another technique of logical preservation is storing provenance data and information of results of the workflow. Retrospective provenance is runtime provenance, so data which is store on execution of the workflow. This retrospective provenance helps in re-execution of workflow. This data however also needs to be specified in a more abstracted way, as it might be too detailed and overwhelming.[10].

Furthermore, Hashan et al. proposed and designed a framework that was based on logical preservation. The main purpose of this framework is to capture detailed information about the execution phase of the workflow in the cloud platform and combine it with provenance data of the workflow. It is recommended that the workflow reproducibility can be ensured by reproviding similar execution using the cloud provenance and then re-execution of the workflow. Even though this method guarantees the re-execution of the scientific workflow, it is not capable of capturing and identifying potential changes to the original workflow[10].

In addition, Belhajjame et al. supported logical preservation by creating a technique named *Research Objects*. This approach focuses on combining and merging different data, so as to increase and support the workflow reproducibility. This data ranges in types from description of the workflow components to provenace data. Howver, this is not enough for ensuring a fully reproducable system, as there is a lack of technical details regarding dependencies and the execution environment.

It can be clearly noticed that these two types of preservations(physical and logical) play a key role in achieving partly workflow reproducibility. Even though, these two approaches individually offer the aforementioned benefits, still they are not sufficient enough to solve the issue of repeatability. On the one hand, physical preservation focuses on recreating the packaged components of the workflow, while their description is absent. As a result, it is an easy task to reproduce the workflow, but it can not reproduce the workflow with different data. On the other hand, logical preservation emphasizes on describing the components of the workflow in detail. However, it is still not enough, as it does not provide information about different dependencies. Therefore, the integration of workflow specification and description of its components alongside a portable packaging mechanism becomes fundamental [10].

4 A Framework for Scientific Workflow Reproducibility in the Cloud

This section gives a notable example of a framework in development by J.C. Rawaa Qasha and P.Watson that tries to increase reproducibility in the cloud. [10] Firstly, the architecture of the proposed framework is described and its main functionalities are mentioned. After

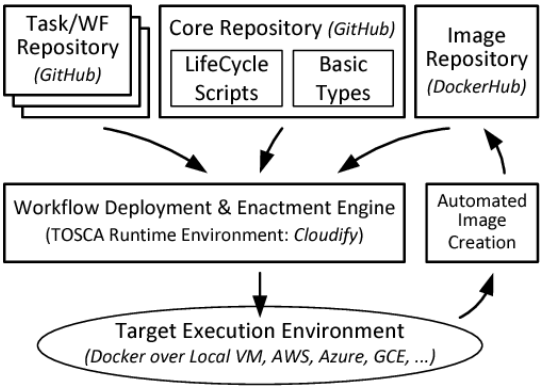


Fig. 2. Model of Reproducibility Framework [10]

that, the operation of the framework is evaluated based on criteria.

4.1 Framework Architecture

The proposed framework is constructed based on four fundamental components: the Core repository, the Workflow and Task repositories, the Image repository supported by the Automatic Image Creation (AIC) facility, and the workflow enactment engine as can be seen in figure 2. The Core repository makes use of a set of common TOSCA (Topology and Orchestration Specification for Cloud Applications) elements such as Node and Relationship Types, and life cycle management scripts. They are crucial components, because they serve as a base for creating not only tasks, but also workflows. The aim of Workflow and Task repositories is to store workflows including their components, which can be shared among different users and act as a foundation for new workflows. The Image repository consists of images, which demonstrate the workflow and its corresponding tasks. The AIC is responsible for capturing automatically these images, which have a positive affect on increasing reproducibility and improving performance of workflow enactment. Finally, the implementation of the workflow enactment engine is enabled by a TOSCA-compliant runtime environment.

The approach of logical preservation is ensured based on the TOSCA specification. TOSCA enables the description of the workflow at the abstract level in combination with the overall software tools, which deploy and execute these workflows. Furthermore, it offers the opportunity to users to deploy and enact the scientific workflows automatically on a different cloud infrastructure.

Another component that is present in the architecture of the framework is a version control platform. This platform is responsible for tracking and managing the changes, which are found in the workflows. In addition, it supports the AIC facility. The AIC ensures the implementation of physical preservations by using Docker. Furthermore, the proposed framework makes use of available source codes such as GitHub. One of the primary benefits of GitHub is that it massively supports the developers, by allowing them to share and improve their individual work, which can help in achieving a better reproducibility. For the readers, who are more interested in having further knowledge on how this framework is constructed and how it functions, you can have a look here [10].

4.2 Framework evaluation

The evaluation of the proposed framework is performed based on three different aspects. The first one is concentrated on the portability of the workflow, which means that it can be executed on different platforms and environments. The second angle is related to AIC, which highly lowers the runtime of a workflow. Finally, the evaluation is mainly focused on the maintaining a workflow reproducibility, even when some of its components change.

- Reproducibility on different Clouds

The aim of the researchers was firstly to test the portability quality of the workflows. Specifically, they tried to re-execute a workflow, which was initially created in a local environment, on three different Clouds and a VM. For this test, they chose four workflows, which differed in structures, number of tasks and the dependency libraries. The chosen workflows were: Neighbor Joining (NJ), Sequence Cleaning (SC), Column Invert (CI) and File Zip (FZ). Further, the scientists had to clone the workflow repositories on four different platforms : a local VM, and Amazon AWS, Google Engine and Microsoft Azure Clouds.

The test proved that the workflows were adaptable to new Clouds, as they were easily re-enacted and provided the same outcome within a similar runtime. Additionally, a mutual development behaviour, where the developers could design and test a workflow locally and then share it among others via Workflow, Task and Image Repositories. This pattern was significantly supported by TOSCA and Docker packaging. For further information of the results, see [10].

- Automatic Image Capture for Improved Performance

The constructed framework allows tasks and workflows to make use of OS images offered by DockerHub, users or the AIC. If a predefined image is used, the need for installing dependency libraries and task artifacts during workflow execution are eliminated. This leads to a considerable decrease of the deployment tasks, which benefits the runtime of a workflow.

The researchers succeeded in proving this theory by running the workflows with different images: the base image available on DockerHub, the base image with pre-installed dependency libraries and task images captured by the AIC. Based on the results[10], there was a significant overhead in using the base image from DockerHub, as the installation of the dependency libraries required less time. The other two options had similar executions, but the workflow that used the image of AIC was slightly better. The main reason is that the AIC captures everything the task needs to run, whereas the second option provides only dependency libraries while the task artifacts were downloaded on-demand.

- Reproducibility in the Face of Development Changes

One crucial approach that can improve the performance of the workflows is to make them embrace potential changes. These changes may influence two layers: the input/output interface of a workflow or task, and their implementation.

The researchers present a hypothetical case. For instance, the users need to save storage space by compressing the workflow output files. In response to that, the developers have to create a new Zip task and add it to the workflow. It has to be pointed out that the changing outcome produced by the workflow changes its interface, as well. The zipped branch of the workflow refers to the Zip/master branch of the task. This reference applies that the workflow will take into account only the latest version of the task in the branch. This is convenient because as the task implementation is improved over time, the zipped workflow will use a tasks latest tagged version. Consequently, the workflows are updated automatically. However, if repeatability has to be strict, the reference to the Zip task would be a specific tag, which will not enable the automatic update of the workflow.

Another considered case was a new available release of a library such as Java, which provides a better performance as its faults are fixed. This available version will notify the workflow to update. The change is compatible with the last version of the workflow and the creation of a new branch is not required. In this case, the changes have to be merged to the required zipped branch.

The researchers consider and other possible scenarios in [10]. By mentioning these cases, they prove that workflows can be adapted to new changes, which assures a better reproducibility(repeatability) of the system.

5 Advantages and Disadvantages of Cloud for workflows

Based on the execution of the last framework in the cloud, this section highlights and describes in details the main benefits provided by this

infrastructure regarding the workflows. There are five significant advantages overall that have been identified[5]. In addition, some of its drawbacks are briefly described.

5.1 Provenance and Reproducibility

In computational science, provenance refers to the storage of metadata about a computation that can be used to answer questions about the origins and derivation of data produced by that computation[5]. It can be considered as computationally equivalent to a lab scientists notebook and it is a fundamental component of reproducibility, the cornerstone of experimental science[5].

Virtualization is an important feature for provenance, as it captures the exact environment that performs a computation, including all of the software and configuration used in that environment [5]. In previous researches, a provenance model was proposed, where the Virtual Machines(VM) played the most important role. The point of this approach is that, if a workflow is executed in a VM, then the virtual machine image can be stored along with the provenance of the workflow. This is a highly significant process, as it gives to the scientists the opportunity to answer a number of concerning questions related to the workflow. Some of these questions would be: "What version of the simulation code was used to produce the data?" or "Which library was used?". In addition, the environment can be re-launched in order to repeat the same experiment or to make modifications and improvements to it.

Another usage of provenance in scientific workflow is automated decision making[8]. By using a provenance model recording values of steps, a quality provenance model, we can make conclusions about the execution process. So can we conclude why a process failed based on the data and can we decide to possibly accept or reject the workflow at an earlier step in the workflow based on the quality of recorded results. Using query languages like "Zoom"[1] custom provenance data can be defined and used for personalized decision making.

5.2 Legacy Applications

As workflow applications usually consist of heterogeneous software components, components that run on different cores, the workflow management system is responsible for coordinating these components in order to create a coherent application. Often this is required to be done without performing changes to the components, as they might affect negatively not only the application, but also the validity of the results.

Clouds and their virtualization technology may make these legacy codes much easier to run. The scientists have the right to install any software tool(e.g. operating systems or libraries) they prefer or the ones that are more suitable for their application and procedures. The resulting environment can be bundled up as a virtual machine image and redeployed on a cloud to run the workflow.

5.3 On Demand

The cloud is able to allocate any available resources on demand. The users can request and be provided with any suitable resources, whenever they need to. If a step in the workflow requires more resources these can be on demand requested. In contrast to local applications, which are limited to their resources up-front, a cloud running workow application can start with only a small amount of the overall needed resources.

However, the drawback of this feature is that, if the needed resources are not made available to answer a user's request immediately, then the request fails.

5.4 Provisioning

Provisioning, the receiving and scheduling of resources, in a cloud are better then grid computing. In contradiction to the grids, where the requested resources are placed in a (probably long) queue and serviced in order, the users in the cloud can request and obtain immediately any desired capability. In this case, they can schedule their resources computations using a user-controlled scheduler. Once the workflow is provided with its needed resources, it directly starts performing its corresponding tasks. This provisioning model offers major benefits to the scientific workflows, as it rapidly increases their performance and drastically reduces the total scheduling overhead.

5.5 Elasticity

Apart from provisioning resources on-demand, clouds give the chance to the users to return the available resources, when they no longer need them. This extra feature of elasticity is a beneficial characteristic for workflow applications, as the workflow systems can simply grow and shrink the corresponding resource pool, based on the needs of the scientific workflow.

5.6 Disadvantages of Cloud for Workflows

Even though cloud offers a number of beneficial features and characteristics towards the scientific workflows, some of its issues need to be addressed[4].

As the scientific workflows are considered as complex systems, building an appropriate environment for the execution of their workflow applications is considerably difficult as these may require many different environment on different sites.

In workflow scheduling, in a cloud environment this most likely will be done over multiple sites. Workflow could be executed from different sites for reasons like, separated data locations or not enough computing resources. The difference between a multi-site and single-site distributed cloud is that a multi-site does not have the same network, bandwidth and resources across the different distributions. For workflows this also brings the added complexity of scheduling and timing execution of steps between different cloud sites. Multisite WMS's could work with (peer to peer) communicating master/slave, schedule activity data being sent to the different master and provenance stored in a provenance database [6], an example of this multi-site setup can be seen in fig.3. However these systems are not optimal as of course this architecture opens up to fault tolerance problems, as failure of one of the master results in failing workflows.

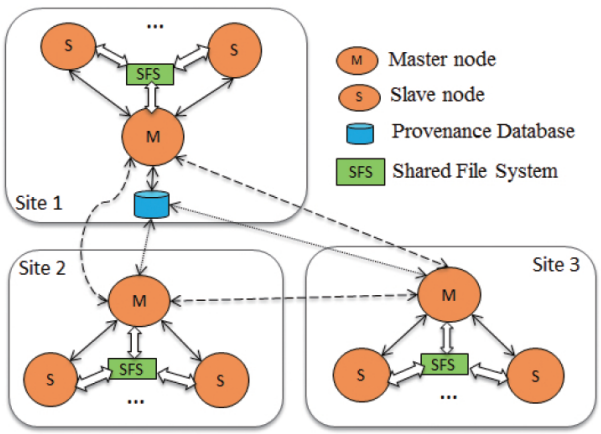


Fig. 3. Multisite workflow system [6]

Another issue is that the IaaS places more system management responsibility on consumers, as they have to manage the virtualized infrastructure and they need to perform system administration.

In addition, another concern is related to security. As the users are allowed to install any legacy software system that fits best their applications, the cloud exposes them to security vulnerabilities of those software systems. Another weakness is that most scheduling algorithms and WMS's are not properly secured.[7] Finally QoS is an important, set requirements should be followed, however most workflow management systems are more focused on execution time and maximum bandwidth.[3] For cloud computing however QoS are very important, as scientist might require different QoS. This makes workflow management systems less fit for cloud environments. As commercial

computing QoS decides pricing, pricing might be a issue for the researcher, while the workflow management system neglects this aspect.

6 Deploying workflows in the cloud

This section presents some methods[5] used for the deployment of the scientific workflows in the cloud. Their main functionalities are interpreted and they are associated with corresponding images.

6.1 Pegasus Worfklow Management System

Workflow Management Systems are mainly responsible for the execution and the monitoring aspect of a scientific workflow. A highly popular example of such a system is Pegasus WMS[2], which can be easily deployed in the cloud. However, this is not the case for all the workflows, as a part of them can not be deployed in the cloud. More importantly Pegasus is even compatible with cloud computing, and handles differently distributed resources. It can manage large workflows(with millions of tasks) and it records data for both workflow's execution and its corresponding results. This type of information is involved in provenance, which can be used to achieve a higher rate of reprodcibility.
Pegasus WMS consists of some core blocks: the Pegasus mapper, the DAGMan execution engine and the Condor schedd responsible for task execution. It adapts workflows to the target execution environment and it manages task execution on distributed resources. In addition, Pegasus WMS provides numerous advantages regarding the workflows. Firstly, it dynamically improves the workflow performance and it is capable of providing the required scientific results much faster. Pegasus WMS additionally ensures highh reliability especially during execution. Consequently, this benefits the scientists and users, as they do not have to deal and fix potential errors and failures constantly. Moreover, they track and record (provenance) data, which can be easily located and accessed during and after workflow execution.

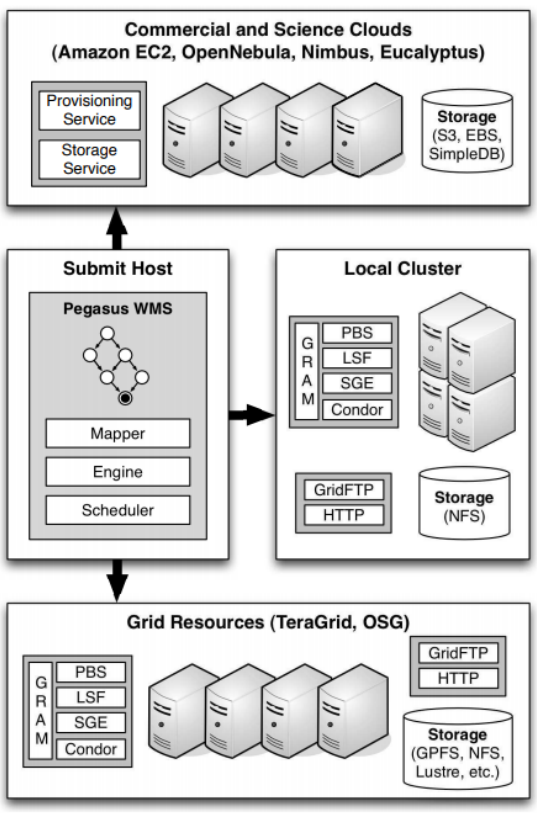


Fig. 4. Pegasus Workflow Management System [2]

6.2 Virtual Clusters

Scientific workflows require huge amounts of compute cycles to process tasks. In the cloud, the virtual machines ensure these cycles are. In order to achieve the best performance possible for large workflows, many quantities of virtual machine instances must be used simultaneously. These collections of VMs are called virtual clusters[5]. The process of deploying and configuring a virtual cluster is known as contextualization[5]. Contextualization consists of complicated configuration stages that might be error-prone to perform manually. For the automation of this process, diffenent software equipment can be used such as the Nimbus Context Broker. This tool collects data regarding the virtual cluster and uses it to make configuration files and start services on cluster VMs. This implies that workflow environments are build up in a similar manner and provenance data is easier to create over what would otherwise be different actual environments, making reproducibility easier.

7 Conclusion

Reproducibility and repeatability is a crucial requirement for the scientific workflows and experiments. It offers to the scientists the chance to verify the validity of their results, to share the proceedings of their experiments and to further develop them. In this paper, we firstly introduce some basic concepts related to our topic. Following that, we mention and describe briefly previous attempt of scientists, who tried to achieve reproducibility. Specifically, a implementation of a framework that supports reproducibility of scientific workflows in the cloud is taken into consideration. The architecture of this framework and its evaluation are provided in details. After that, advantages and disadvantages of the the clpud are examined, with an emphasis on why cloud is a good solution to reproducibility. Then, some deployment methods of the scientific workflow applications are displayed and explained.

Scientific workflow and workflow management in cloud environments, is a fast growing and changing field, offering more features and functionality through tools and algorithms like for example in paper earlier mentioned Pegasus, Zoom, DIM. However in this field there still is a lot to be improved and developed. So there is still security and fault tolerance problems, aswell as high cost in managing and configuring workflows and is reproducibility hard to achieve in complex systems.

8 Future work

As science and technology are rapidly evolving, new efficient solutions and approaches will be probably discovered for the execution of the scientific workflows. This means that for scientific workflows, because of increasing complexity, new development should also be kept up in the area of reproducibility.

For scientific workflows in respect to cloud computing, future researches will possibly focus on the PaaS and SaaS clouds, which will be likely developed exclusively for the workflow applications. For instance, a PaaS cloud may provide a user-centered Replica Location Service (RLS) for locating files in the cloud and outside, a dynamic Network Attached Storage (NAS) service for storing files used and created by workflows, data such as provenance data.[5, 6] A SaaS cloud for workflows may provide an application-specific portal where a user could enter the details of a desired computation and have the underlying workflow services generate a new workflow instance. In addition, future research might cover as a topic the integration of the grid and the cloud for the deployment of the scientific workflows[5]. In respect to provenance, we expect development in the area of more customizeable provenance data as offered by tools like Zoom[1], to increase ease in reproducibility and decision making[8] .

Acknowledgements

The authors wish to thank our expert reviewer Dimka Karastoyanova and our colleagues for their valuable feedback.

Author	Subject	Tools	Includes Cloud	Provenance relation
Cohen-Boulakia et al., 2008	Provenance	Zoom	No	Storing custom defined provenance data using zoom.
Deelman et al., 2015	WMS	Pegasus	Yes	-
Gil et al., 2007	Reproducibility & Provenance	-	No	Difficulties of provenance and reproducibility in scientific workflows
Juve, 2012	WMS	Pegasus	Yes	-
Liu et al., 2016	Provenance & Scheduling & WMS	Chiron	Yes	Supporting provenance in multi-site workflow scheduling
Masdari et al., 2016	Scheduling & WMS	-	Yes	-
Missier et al., 2008	Provenance	VisTrail & Zoom	No	Automated workflow decisions based on provenance
Rawaa Qasha & Watson, 2016	Reproducibility & Provenance	Git	Yes	A framework for improving reproducibility in workflows with multiple repositories

Table 1. Summary of used sources

REFERENCES

[1] S. Cohen-Boulakia, O. Biton, S. Cohen, and S. Davidson. Addressing the provenance challenge using zoom. *Concurrency and Computation: Practice and Experience*, 20(5):497–506.

[2] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger. Pegasus: a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015. Funding Acknowledgements: NSF ACI SDCI 0722019, NSF ACI SI2-SSI 1148515 and NSF OCI-1053575.

[3] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers. Examining the challenges of scientific workflows. *Computer*, 40(12):24–32, Dec 2007.

[4] G. Juve. Scientific workflows in the cloud. University of Southern California.

[5] G. Juve and E. Deelman. *Grids, Clouds and Virtualization*. Springer, 2011.

[6] J. Liu, E. Pacitti, P. Valduriez, and M. Mattoso. Scientific Workflow Scheduling with Provenance Support in Multisite Cloud. In *VECPAR*, page 8, Porto, Portugal, June 2016. Faculty of Engineering of the University of Porto, Portugal.

[7] M. Masdari, S. ValiKardan, Z. Shahi, and S. I. Azar. Towards workflow scheduling in cloud computing: A comprehensive analysis. *Journal of Network and Computer Applications*, 66:64 – 82, 2016.

[8] P. Missier, S. Embury, R. Stapenhurst, J. Freire, D. Koop, L. Moreau, S. Imaging Inst, and U. Utah. Exploiting provenance to make sense of automated decisions in scientific workflows. In J. Freire, D. Koop, L. Moreau, S. Imaging Inst, and Utah, editors, *2nd International Provenance and Annotation Workshop*, volume 5272, pages 174–185, Germany, 2008. Springer. Microsoft Corporat, Univ Utah Sci Comp Missier, Paolo Embury, Suzanne Stapenhurst, Richard 11 BERLIN BIU90.

[9] J. Qin and T. Fahringer. *Scientific Workflows- Programming, Optimization and Synthesis with ASKALON and AWDL*. Springer, 2012.

[10] J. C. Rawaa Qasha and P. Watson. A framework for scientific workflow reproducibility in the cloud. *IEEE 12th International Conference on e-Science*, 2016.

Predictive monitoring for Decision Making in Business Processes

Ana Roman, Hayo Ottens

Abstract— The achievement of business goals for any service or establishment is always desired, whether it concerns for example the health of patients in hospitals, the maintaining of a company image or the making of profit in profit making organizations. These business goals are achieved by (series of) business processes. In order to achieve these goals, the processes should be monitored at runtime, but their outcomes can also be predicted to prevent unwanted scenario's. Instead of monitoring business processes upon completion, it is desired to intervene before a sub-optimal action is taken. Possible options and their risks should be taken into account in order to make a well-informed decision.
Information systems currently used to support business processes keep their historical data in the form of logs. These logs can be used as a means to hint towards a more optimal action in a certain situation where a decision is to be made.
In this paper we present multiple methodologies to 1) predict the outcome of a (sub)process and 2) give a risk-analysis for possible decisions to take. For the purpose of process execution, the business goals and constraints can be defined in the form of a model. Next a framework is used to monitor and identify the input data values which can cause the business goal to be unreachable. We will elaborate on using existing outcome-oriented predictive business process monitoring techniques as well as a way to predict the risks from log-data. Lastly, we will point out weaknesses of these methods and present research questions for future research.

Index Terms—Business processes, Decision making, Predictive monitoring

1 Introduction

The goal of this this paper is to point out requirements for a system that is capable of predictive monitoring for decision making in business processes, as well as to list some of the used techniques, methods and frameworks. Moreover, we try to find the weaknesses and topics for further research in this area. A *business process (BP)* is a summation of events, activities, and decisions which lead to a certain outcome that is of value to the involved actors [11]. *Business process monitoring (BPM)* is a technique that provides the means to manage the ongoing execution of a process, in order to understand its behavior and its compliance to a set of business goals[8]. BPM is not meant to improve the ways in which activities are performed, but rather to manage entire chains of events, activities and decisions and to ultimately add value to the organization and to its customers [4]. *Predictive (business) process monitoring* techniques build on top of the conventional ones by allowing the prediction of future states or outcomes of a business process. Examples are: the approximate remaining execution time or the probability of a certain business goal to be achieved [8].

In recent years, a lot of research has been aimed towards predictive monitoring methods - also named *outcome-oriented predictive process monitoring methods*. In our study, we will take a look at how those are being carried out, we will analyze them and at the same time, take a look at the risks involved when using those methods.
We will try to answer the following research question: how can predictive monitoring methods be combined with risk monitoring methods in the context of business processes?

The paper is organized as follows: First, components of predictive business process monitoring methods are listed and briefly explained. Next, we go into detail on the requirements of any predictive monitoring system as well as point out several techniques and frameworks. In section 5 we will shortly explain the method used for our research. The actual comparison of techniques is then evaluated in section 6. Lastly, we give a summary in section 7 and in section 8 we are discussing our view on the results as well as give perspective to future research.

- *Ana Roman, E-mail: ana.ro628@gmail.com.*
- *Hayo Ottens, E-mail: h.m.ottens@student.rug.nl.*

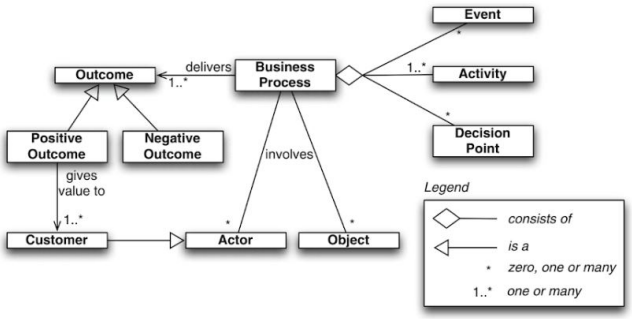


Fig. 1. The components of a business process, taken from [4]

2 The components of (predictive) business process monitoring methods

Every business process includes a number of *events*: actions that happen atomically, without duration and *activities*: actions that can be triggered by an event. For example, the placement of a business order is an event. This event may trigger the execution of a series of activities, such as, the inspection of the order by the manager and the execution of tasks required to fulfill it. These tasks are activities, due to the fact that they require time to be fulfilled [4].
In addition to events and activities, processes typically include *decision points* - temporal points where a decision is made which influences the way in which the process is being executed. A process also involves *actors*: they can be either human actors, organizations or systems that are acting on behalf of human actors; *physical objects* such as equipment, materials, documents, products; *informational documents* such as electronic documents or records [4].
The execution of a process leads to one or several *outcomes*. In an ideal scenario, the outcomes of a process should deliver value to the actors involved. In some situations, this value is not achieved or is only partially achieved, or might not even be gained - this situation corresponds to a *negative outcome*, which is opposed to the *positive outcome* that delivers the most value to the actors.
Figure 2 is meant to illustrate the relationship between the components of a business process.

3 Requirements of a predictive monitoring system

In order to build a system which is capable of predictive monitoring of business processes, we need to fulfill a set of requirements as listed in the research of [2]:

R.1 Define monitoring points and expected behaviour.

R.2 Capture and process the information required for monitoring.

R.3 Normalize the information captured.

R.4 Process event and data information.

R.5 Identify and notify problems.

R.6 Develop automatic support.

3.1 Making the model

Making a model is the first step necessary for making predictions about the outcome of future events [2] and addresses requirement **R.1**. Multiple ways exist to obtain such a model. This process model should include all data attributes of sources which need to be monitored.

A complete understanding of the process is hard to define. Different process participants may need different aspects of the process for analyzing process events from their respective perspectives. Moreover, the amount of data and data heterogeneity is abundant, e.g. messages, event logs and many other artifacts in many different formats [1].

Currently, there are multiple existing formats available for modelling these events. One of them is YAWL (Yet another Workflow Language) [6]. In the paper of [3], this management system is used to build a model which is used to make predictions about the future. Another approach is Linear Temporal Logic (LTL) as used in [8]. LTL is a modal logic with modalities devoted to describe time aspects [10]. Classically, LTL is used in the literature for expressing business constraints on procedural knowledge [9]. In the paper of [8], LTL is also used in this way.

The above two approaches work well with existing BPM frameworks, further elaborated in 3.2. When a custom framework is desired, the research of [1] gives a useful approach to build a model from scratch. Beheshti et al. introduce in this paper a "graph-based data model for modeling the process entities (events, artifacts and people) in process logs and their relationships, in which the relationship among entities could be expressed using regular expressions" [1].

3.2 Data acquisition

The next requirement is to capture and process the information required for monitoring, as stated in **R.2**. This act can be referred to as 'process mining' [12]. Since nowadays, many different information systems, e.g. Workflow Management Systems (WMS), Customer Relationship Management (CRM) systems and Enterprise Resource Planning (ERP), are in use within organizations which support the execution of business processes [5], we need a way to capture them in ideally one place to be able to monitor these processes. Typically, all these systems are able to provide 'event logs', i.e. a form of registration of what has been executed in the organization. These logs contain a variety of data, for example the process instance (case), dates and times and people involved.

One extensible process mining framework is ProM. This framework is used in the paper of [8], where a prediction and recommendation module is built.

Whereas existing frameworks like ProM may do most of the tedious work already, it should not go unnoticed that capturing and processing the information are required for monitoring, as stated in **R.2**.

In figure 2 we see an overview of the steps in analyzing process event logs (i.e. preprocessing, partitioning, and analysis) as used for the custom model of [1]. In order to achieve this, Beheshti et al. created an extension of a widely used graph-based querying language SPARQL, FPSPARQL, where subgraphs can be constructed and retrieved and transitive relationships are first class objects, as desired. This way, they handle both requirements **R.2** and **R.3**. In their discussion they conclude a great performance of their querying language, even compared to the one used in well-known HyperGraphDB. They conclude though, that the quality of discovered transitive relationships is highly related to the regular expressions generated to find patterns in the log.

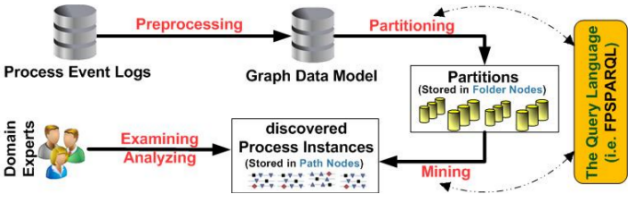


Fig. 2. Event log analysis scenario, taken from [1]

3.3 Detecting anomalies

The following subsections are meant to describe the topic of anomalies in different types of business processes, as well as how they can be handled or prevented.

3.3.1 Regular task monitoring

Being able to detect anomalies when they happen is valuable, because the process can be stopped when the anomalies are detected, and necessary measures can be taken in order to handle them. Performing task monitoring on the execution of a process enables us to supervise the evolution of the process and to check whether this is performed correctly or not. At the same time, this allows the detection of these possible anomalies in the behavior of the process. The status of the execution is obtained by analyzing the events that take place alongside the task. Anomalies are detected by values exceeding their thresholds, for example time, space or some budget running out.

However, we can take into consideration one other approach, that could improve even more the outcome of the process. That is, the ability to prevent the thresholds from being reached. Thus, we can use prediction to avoid the thresholds from being reached, and implicitly the anomalies from appearing.

3.3.2 Predictive task monitoring

As stated in **R.5**, the system needs to be able to identify problems and notify actors about them before an undesired outcome will occur. Given a set of incomplete business processes, predictive process monitoring aims at predicting its class label (expressing its outcome according to some business goal), by studying from a set of completed cases with their known class labels [11]. In the research of [11], Teinemaa et al. aims to answer the following question: Given an event log of completed business process execution cases and the final outcome (class) of each case, how to train a model that can accurately and efficiently predict the outcome of an incomplete (partial) trace, based on the given prefix only? This overarching question is then decomposed into the following sub-questions:

Q.1 What methods exist for predictive outcome-oriented monitoring of business processes?

Q.2 How to categorize these methods in a taxonomy?

Q.3 What is the relative performance of these methods?

Subsequently in the paper, Teinemaa et al. try to answer these questions by using the methods of their found literature on a set of datasets. In the next paragraphs, we try to give an overview of this research. For the full details, please refer to [11].

3.3.3 Taxonomy

In their research, Teinemaa et al. found that there are two main phases, namely the *offline phase* and the *online phase*. The offline phase is determined to train a model in order to next make predictions about running cases, that is in the online phase. The offline phase consists of four steps. The first step is to use the event logs to extract the case prefixes (event identifiers) and to filter them appropriately. Secondly, the prefixes are divided into so called buckets by similarities among the prefixes. This is also referred to as 'hashing'. Next, features are encoded from the buckets to prepare for classification. The final step is to actually train the classifier with the encoded prefixes. Then, the next phase commences.
In the online phase, the goal is to predict the outcome of a running case by reusing the previously determined elements. For this phase, the correct bucket is determined given the running process sequence (trace) and the set of buckets of historical prefixes. This information is then used to encode the features of the running trace for classification. Finally, a prediction is extracted from the encoded trace using the correct classifier for the determined bucket. The steps of these two phases are visualized in Figure 3 (offline) and 4 (online) and explained in further detail in 3.3.4. The steps of these phases form the answer to **Q.2**.

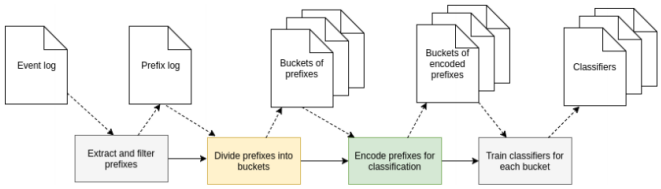


Fig. 3. Offline predictive process monitoring workflow, taken from [11]

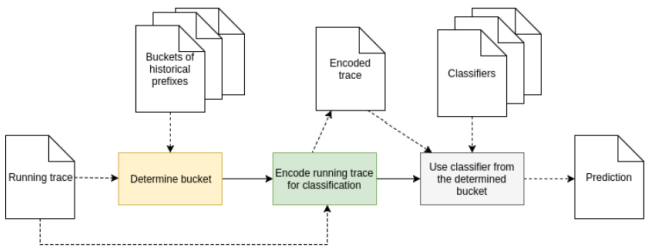


Fig. 4. Online predictive process monitoring workflow, taken from [11]

3.3.4 Existing methods

The phases with their corresponding steps are a global approach to tackle the predicting problem. According to the research of Teilemann et al. different studies use a different implementation of these steps.

- **Prefix extraction and filtering**
  To assure the training data is comparable to the testing data, the prefix log is used for training. It appears though, that using all possible prefixes leads to slowing down the training process and generating a bias. Two different approaches to tackle these problems are found in the study [11]: the *length-based* and the *gap-based* approach. The length-based approach considers prefixes up to a certain number only, whereas the gap-based approach considers every prefix who's length is that of a base number plus every n^{th} number, where n equals the gap.

- **Trace bucketing**
  Teilemann et al. found that typically, the prefix traces in the event log are divided over several buckets, as said in 3.3.3, and different classifiers are then trained for each such buckets. In the online phase, the most suitable bucket is selected and its respective classifier is applied to be able to make a prediction. The following bucketing approaches are used in the studies: Single bucket, KNN, State, Clustering, Prefix length and Domain knowledge. For an explanation of these approaches, please refer to [11].

- **Sequence encoding**
  A fixed length of feature vectors is needed for any classifier to train. It appears to be a real challenge to maintain the same number of feature vectors while each executed event reveals additional information. The trace-abstraction technique solves this problem by considering for example the last n event of a trace. This raises the problem though, that a balance needs to be made between generality and loss of information, which appears to be a difficult task. The next step after choosing a trace-abstraction is to apply a feature extraction function on each event data attribute. The following feature extracting approaches are used in the studies: Static, Last state, Aggregation and Index and are enlisted in figure 5. For an explanation of these approaches, please refer to [11].

Encoding name	Relevant attributes	Trace abstraction	Feature extraction	
			Numeric	Categorical
Static	Case	Case attributes	as is	one-hot
Last state	Event	Last event	as is	one-hot
Aggregation	Event	All events, unordered (set/bag)	min, max, mean, sum, std	frequencies or occurrences
Index	Event	All events, ordered (sequence)	as is for each index	one-hot for each index

Fig. 5. Encoding methods, taken from [11]

- **Classification algorithm**
  The last step in the taxonomy is predicting the outcome from the encoded trace using the correct classifier for the determined bucket. Multiple classification algorithms are being used or experimented with, as found in the research of Teinemaa et al. Two popular ones are decision trees (DT) and random forests (RF). Also support vector machines (SVM), generalized boosted regression models (GBT), gradient boosted trees (XGBoost) and logistic regression (logit) are used.

Please refer to [11] for a more detailed explanation of these methods.

3.3.5 Methods' performance

As seen in section 3.3.4, the taxonomy consists of different steps where in each step multiple techniques and methods are applicable. The research of Teinemaa et al. tries to compare different combinations of methods in the offline phase as well as different combinations in the online phase. The performance of these method-combinations is measured by using running time and accuracy among other things. Method combinations are indicated by using abbreviations coupled with an underscore ('_'). In this section we aim to give a short overview of the performance of these methods. For the complete list of results, please refer to [11].

- **Accuracy**
  Accuracy is measured by the Area Under Curve measure (AUC) and F-score.
  Online, the XGBoost classifier has the best performance with the highest AUC in 15 of 24 datasets and the highest F-score in 11 datasets. Next, RF has the highest AUC in 11 datasets and the best F-score on 14 datasets. Then, Logit has the highest AUC in 7 datasets and the highest F-score in 6. SVM performs significantly worse than the other algorithms, with just two datasets as an exception.

The best performing bucketing and encoding methods are singe_agg with the best AUC in 10 of 24 datasets and 9 best F-scores. Next, prefix_agg performs best with the highest F-scores on 8 datasets, followed by cluster_agg, state_agg, and prefix_index. Prefix_laststate, knn_laststate, and knn_agg are found to perform significantly worse than prefix_agg.

- **Running time**
  The running time of the algorithms are obviously influenced by the size of the datasets. Overall the logit classifier seems to be the fastest and RF slowest. In smaller datasets, SVM is second, followed by RF and XGBoost. In larger datasets though, XGBoost scales better, whereas SVM is slowest.
  Concerning the bucketing and encoding methods, the fastest seems to be prefix_laststate, which is faster in 17 of 24 datasets. Next, knn_laststate, state_laststate, and single_laststate follow up. In the offline phase, in general, the most time is taken by index-based encoding to construct the sequences of events. The fastest bucketing approaches are state-based, then prefix-length based or clustering based, followed by single bucket as the slowest. This could mean that training multiple small classifiers (trained on a subset of the data) performs faster than training a few larger classifiers.

4 Risk management in BPM

Risk management is a research area with implications in various fields. The analysis of risks is to be taken into consideration whenever there is talk of business processes, since failing to address those risks or not having ways to mitigate them may lead to substantial financial losses, reputational consequences and faults that might threaten the organization's existence.
Decision making theory defines risk as *the reflecting variation in the distribution of possible outcomes, their likelihoods and their subjective values*[7]. A *process-related risk* measures the probability of a negative outcome (also called a *fault*), and the way this will impact the process objectives [3].
Risk-aware Business Process Management tries to model risks before the start of the processes, it tries to detect them as early as possible during execution and helps with the mitigation of the risks by taking decisions about the process instances. But there is a certain limitation with this approach, such that in this way, risks are not prevented but rather taken into consideration when the probability exceeds a certain threshold value. This is not always ideal, since the costs of risk mitigation may not always be acceptable, and moreover, risk mitigation is not always possible for all the process risks. Therefore, the previous research [3] by Conforti et al. proposes a method that is meant to reduce the presence of risks; for every process, the method should return the likehood and severity of the occuring of a fault, obtained via a *function estimator*. The function estimator is a trained model based on historical data, as extracted from process logs.

The suggested method from literature

In the previous work of Conforti et al, a method is described that can aid the decision making before the execution of a business process, with regards to the process-related risks.
In order to implement this new method, the four phases of the BPM lifecycle (Process Design, Implementation, Enactment and Analysis) are extended with elements that make up risk management processes.

1. *Risk Identification* is a new initial phase that takes place before the Process Design. It is during this phase that possible risks, faults or negative outcomes are identified as potentially appearing during the execution of the business process.

2. The faults and their risks that are detected in the initial phase are then modelled together with the business process during the Process Design phase, such that the model that is obtained is a risk-annotated process model.

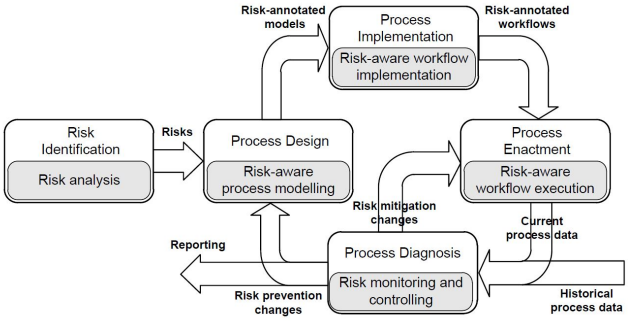


Fig. 6. Risk-aware BPM lifecycle, taken from [3]

3. Later in the process, the risks and their faults are linked to specific aspects of the process model, such that a detailed mapping is created.

4. In the Process Enactment phase, the risk-annotated model is executed.

5. The information obtained during the Process Enactment phase is analyzed together with historical data during the Process Diagnosis phase. The occurrence of risks is monitored, which may result in mitigation in order to recover the process from a fault. Figure 4 is meant to depict how these phases take place.

5 Methods

Our research is based on the findings of work previously done on the subject by different research groups. The literature studied in this paper is partly provided by our supervisor, Miss Karastoyanova, and partly by literature research of ourselves. The found literature provides insight on different aspects of the subject. Our research method is the following: in order to write this paper, we carefully put together different ideas from the literature in the most chronological order. The aim of this approach is to provide the basis of information that can be used to build a system for predictive monitoring, designed according to the previous findings from the literature.

6 Results

As our aim of this paper is to give an overview and comparison of currently used techniques and frameworks, this section will be used to briefly go over these previously addressed techniques and frameworks. Firstly, let us remember that there are different stages of predictive monitoring: making a model, data acquisition, classifying anomalies and predicting process outcomes.

6.1 Making a model

In section 3.1 we found three different methods on defining a model for predictive monitoring. The most tedious but custom approach is to build the model yourself. The research of [8] does so by using their custom graph-based modelling language 'FPSSPARQL'. One alternative is YAWL (Yet another Workflow Language). It is a workflow management system and has a powerful process modelling language that supports all control-flow patterns. Linear Temporal Logic (LTL) on the other hand is a constraint specifying tool. Both of these approaches can go hand in hand and work well with existing BPM frameworks.

6.2 Data acquisition

Section 3.2 covers the techniques used to acquire data from multiple sources. ProM is in the literature denoted as a widely used platform-independent framework to do this. It supports a wide variety of process mining techniques in the form of plug-ins. The research of [8] defines a way to build a custom data acquisition system. The quality of the data is highly related to the regular expressions used to acquire the data, as they argue.

6.3 Detecting Anomalies Through Predictive Monitoring

In section 3.3 we go into detail of the existing methods for predictive monitoring of business processes. By looking at the previous research [11] of Teinemaa et al., four steps are studied that aid the methods: prefix extraction and filtering, trace bucketing, sequence encoding and lastly finding the correct classification algorithm. These steps are then tested in an experimental set-up, and all the algorithms are investigated with regards to time performance and accuracy.

6.3.1 Accuracy results

In terms of accuracy, the study has concluded that the best performing classifier is XGBoost, followed by RF. The difference between the three performances are not statistically significant, according to the paper. The worst accuracy performance is achieved by SVM.
The choice of choosing the sequence encoding had a greater effect on the results than the bucketing method, such that, the best results are mainly achieved using the aggregation encoding with either the single bucket, clustering, prefix length based or state-based bucketing. These methods generally achieve comparable results.

6.3.2 Temporal results

In the offline phase of the process, the classifier that had the best time performance is logit. When dealing with smaller datasets, the second best performance is achieved by SVM, RF and then XGBoost, although XGBoost seems to scale better than the others in the large datasets. In the offline phase, RF is the slowest classifier, while the other three have yielded comparable results. When it comes to bucketing, state-based bucketing is found to be the fastest approach in the offline phase, followed by either prefix length or the clustering base method, and the slowest being single bucket.
Teinemaa et. al then proceed by using the XGBoost classifier, and find that in general, the time for applying the aggregation functions is small compared to the time that is needed for training the classifiers.

The conclusion of the research is that the use of an aggregated encoding of the sequence of activities give the most reliable and accurate results, due to the fact that this type of encoding allows the representation of all prefix traces in the same number of features.

7 Conclusion

The goal of this this paper is to point out requirements for a system that is capable of predictive monitoring for decision making in business processes, as well as to list some of the used techniques, methods and frameworks. Moreover, we try to find the weaknesses and topics for further research in this area. In the preceding sections we gave an overview of the required steps and components of building a predictive monitoring system. The research of [2] pointed out the following requirements: Define monitoring points and expected behaviour, Capture and process the information required for monitoring, Normalize the information captured, Process event and and data information, Identify and notify problems, Develop automatic support. In section 4 we specify about risk management and we extend the four phases of the BPM lifecycle with risk management processes. Lastly, in section 6 we list all used methods in every step of building a predictive monitoring system and try to compare them. We have seen that multiple tools and frameworks can work together to form the basis of such a system: YAWL, LTL and ProM. Then XGBoost is found to be the best classifier in terms of accuracy, whereas the logit classifier performs best timewise.

8 Discussion

The present paper aims at putting together methods for predictive monitoring based on past results of different research groups. By taking into consideration different aspects of such results, we describe the important characteristics of the methods.The novel idea that this paper proposes is the complete predictive method, together with risk analysis and anomalies detection. While these three aspects have been researched before, there is no previous study that researched this topic with all three aspects in mind. Thus, our paper aims to work with the already existing methods and to put them together, in order study them as a whole and not as independent methods.
To note is also that our team did not conduct any independent research and neither have we obtained results of our own, since that is not the purpose of the paper.
The methods that are described in the present paper have not been investigated in real life or experimented with, thus we shall leave this topic open for future research and study. In order to analyze the efficiency of our study and to measure whether the application of our method on real life cases does indeed have a positive outcome, the method should be applied to a real business case that is monitored from start to finish.
Another topic that our research does not dive very deep in is the classification of anomalies and different methods of how they can be treated based on this classification. For example, temporal anomalies should be treated differently from monetary anomalies, and so on. We also leave this open for future research.

Acknowledgements

The authors wish to thank especially Miss Karastoyanova for carrying the task of supervisor. Also, we would like to thank Ruben Scheedler and Derrick Timmerman for their reviews. And lastly, we acknowledge the hard work of the previous researchers whose papers have made possible the completion of our study.

References

[1] A. Beheshti, B. Benatallah, H. R. Motahari Nezhad, and S. Sakr. A query language for analyzing business processes execution. volume 6896, pages 281–297, 01 2011.
[2] C. Cabanillas, C. Di Ciccio, J. Mendling, and A. Baumgra. Predictive task monitoring for business processes. 09 2014.
[3] R. Conforti, M. de Leoni, M. La Rosa, and W. M. P. van der Aalst. Supporting risk-informed decisions during business process execution. In C. Salinesi, M. C. Norrie, and Ó. Pastor, editors, *Advanced Information Systems Engineering*, pages 116–132, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.
[4] M. Dumas, M. Rosa, J. Mendling, and H. Reijers. *Fundamentals of Business Process Management*. Springer Berlin Heidelberg, 2018.
[5] M. Dumas, W. M. van der Aalst, and A. H. ter Hofstede. *Process-aware Information Systems: Bridging People and Software Through Process Technology*. John Wiley & Sons, Inc., New York, NY, USA, 2005.
[6] A. Hofstede, ter, W. Aalst, van der, M. Adams, and N. Russell. *Modern business process automation : YAWL and its support environment*. Springer, Germany, 2010.
[7] B. Levitt and J. G. March. Organizational learning. *Annual review of sociology*, 14(1):319–338, 1988.
[8] F. Maggi, C. Di Francescomarino, M. Dumas, and C. Ghidini. Predictive monitoring of business processes. 12 2013.
[9] M. Pesic, H. Schonenberg, and W. M. P. van der Aalst. Declare: Full support for loosely-structured processes. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 287–287, Oct 2007.
[10] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)(FOCS)*, volume 00, pages 46–57, 09 1977.
[11] I. Teinemaa, M. Dumas, M. L. Rosa, and F. M. Maggi. Outcome-oriented predictive process monitoring: Review and benchmark. *CoRR*, abs/1707.06766, 2017.
[12] C. Turner, A. Tiwari, R. Olaiya, and Y. Xu. Process mining: From theory to practice. *Business Process Management Journal*, 18, 06 2012.

A Comparison of Peer-to-Peer Energy Trading Architectures

Anton Laukemper and Carolien Braams

Abstract—Peer-to-peer energy trading (P2P DET) enables people to trade their own generated energy from renewable energy sources (RES) among end users in a distribution network. These end users who both produce and consume energy are called prosumers. The rise of RES like photo voltaic panels and wind turbines presents numerous opportunities; for example, the risk of power outages is reduced through multiple redundant power sources. However, it also poses technological challenges in the design of electric grids considering issues such as privacy, security and reliability of power supply. Recent research has shown various methods to address these challenges by finding new techniques to balance supply and demand of electricity, and to transmit it from prosumer to prosumer using a smart grid. This paper gives an overview of the various aspects that come with a distributed grid of power producers and compare approaches of recent works. We point out that a possible architecture of a distributed peer-to-peer smart grid consists of two parts, a system that dynamically responds to variations in demand, and a system that securely and reliably sends power packages through the grid. The main contribution of the paper is a qualitative comparison of a selection of possible P2P DET platforms. Although many of these platforms have similarities, The comparison shows that the platforms differ in the grid structure that underlies the platform, mainly because they focus on different subproblems of P2P energy trading.

Index Terms—Smart grid, Peer-to-Peer energy trading, Power Routing, Demand Response Optimisation Models, Microgrid, Renewable Energy Sources

1 Introduction

The production and distribution of electrical energy is commonly organized in a centralized manner; The current power supply system is a rather one-directional, hierarchical system in which few big producers deliver energy over a network of connected transmission lines, the power grid, to multiple consumers. [8]. An electrical energy grid has two characteristics. The first characteristic is that the frequency and phase of the electrical current need to be synchronized on each of the grid's distribution levels [2]. Secondly, the power supply must match the demand at all times in a 'just-in-time paradigm' [8].
The increasing number of renewable energy sources (RES) such as wind turbines and photo-voltaic units poses a significant problem to such a grid. In contrast to conventional power plants, whose output can be carefully controlled, RES's energy production is unpredictable and uncontrollable. This makes the power flow management that guarantees synchronization of the grid and a balance of supply and demand exceedingly difficult.
Due to the expansion of distributed RES, a newer and smarter structure of the energy grid, called a smart grid, is likely to hold the future. This promising design comes in the form of smart micro-grids; a partitioning of the main grid into smaller asynchronous cells that are able to exchange energy through "digital grid routers" [2]. Together with increasing energy storage capacities, smart grids can facilitate a large scale penetration of RES in the power grid. Since individual households are able to install RES, a completely new market arises in which traditional consumers can also produce and sell surplus electricity. All actors in the smart grid can therefore exchange energy to one another. The end users that both produce and consume energy are called "prosumers".
This peer-to-peer (P2P) distributed energy trading (DET) offers a multitude of advantages. It raises a whole new energy market between end users of the grid and by this, eliminates the importance of the utility companies. The Peer-to-peer distributed energy trading (P2P DET) will be an important element for the exchange of energy in the new power grid.
Nevertheless, P2P DET in a smart grid poses also completely new challenges. Firstly, within a smart grid cell, the balance between supply and demand must still be maintained, though there would be many more energy producers, supplying power at unreliable and unpredictable times. The dynamical response to variations in demand is called "load matching". A second challenge entails that energy would not only be transported in one direction from power companies to consumers, but also from prosumers to other prosumers, and therefore the grid must support a bidirectional flow of energy from each prosumer to everyone else. This makes the grid vulnerable to security and privacy threats, that have to be addressed in possible implementations. The process of sending power packages securely and reliably sending through the grid is called "power routing" [1].

There have been many different surveys on the topic of P2P energy trading in the past. The study performed by Abdella and Shuaib [1] presents the current state of research on power routing and gives an extensive discussion of load matching approaches. Saputro, Akkaya and Uludag [8] classify and review a number of power routing algorithms, while Vardakas et al. [12] do the same for different load matching approaches. Chenghua Zhang et al. [14] compare different existing peer to peer Energy trading projects.
Neither of these studies, however, give a detailed comparison of concrete DET platforms that manage to solve, at least partly, the main challenges of P2P DET, namely load matching and power routing. Therefore, we try to illuminate the connection between both parts of P2P DET. This paper aims, among other goals, to facilitate a better understanding of the different components of an energy trading smart grid.
Section 2 will lay out the details of the two main challenges of P2P DET. Section 3 discusses the focus points we want to review by examining different P2P DET architectures. In section 4, we present four different possible implementations of DET platforms. We illustrate in what way they address the aforementioned challenges. Furthermore, we examine each platform regarding different criteria, such as whether privacy and security issues are considered, and what energy storage concepts they take into account, and compare them in section 5. Section 6 discusses the findings of the survey and possible future work. Finally, section 7 concludes the research presented in this paper.

2 Load matching and Power Routing

This paper addresses two challenges that are key elements for succeeding P2P DET, namely load matching and power routing. The following subsections clarify the meaning of these concepts in detail.

- *Carolien Braams is a first year Computing Science master student at the University of Groningen.*
- *Anton Laukempers a first year Computing Science master student at the University of Groningen.*

2.1 Load Matching

If consumer demand during peak hours extends the energy production levels in the power grid, system failures and black outs can follow. Intuitively, one might suggest to turn up more power generating units during peak hours to cover the extra demand. In practice however, this comes with high operational costs because the reserve power units are most of the time underutilized and are expensive to boot up [1].
Power companies therefore rather use strategies that influence the consumers energy usage behaviour in such a way that their total power consumption is reduced, benefiting the consumer, but also trying to make the demand match the supply. Strategies like this fall under the term "demand side management" [12].
Demand side management is a long-term strategy that aims to decrease power consumption at every hour, e.g. through automatizing heating and ventilation. A specific subset of programs that especially aims at making consumers change their behaviour on the short-term when the grid's stability is in danger, is called "demand response" (DR) [12].
Methods to perform demand response include "direct load control", which gives the utility company the control to shut off certain appliances when needed in exchange for a financial benefit and "demand side bidding", where consumers can make offers to reduce their load at certain times, also in return for a specified financial benefit. A third method is to increase the energy price during peak hours which indirectly gives the incentive to reduce the load.
The introduction of smart grids, RES and peer-to-peer energy trading offers completely new opportunities but also complications to this task. On the one hand, smart appliances, smart metering systems and energy storage systems facilitate DSM and enable consumers to switch to local energy production or stored capacities during peak hours, on the other hand, it is much more difficult to synchronize demand and supply with increasingly fluctuating energy production.

2.2 Power routing

To accomplish a smart grid that enables P2P trading, three conditions must be met. First, to be able to actually transport energy from one prosumer to the other, the grid must support some kind of address system. Secondly, it must be able to transport power in both directions, because prosumers can both sell and demand energy at different times. And thirdly, it must be able to convert energy from RES that is produced in DC to AC and synchronize the frequency and phase of the current with the local grid cell.
The process of sending power through a grid under these conditions is called power routing. Since energy is lost if it is transported over long distances, the main challenge is to find the optimal route from A to B. Many researchers [8, 9, 10] have suggested to view power routing similar to data packet routing as it is done on the internet. Electricity would then be packaged and send to the receiving *Power Router* [8], which is identified by IP address. Therefore, power routing can be seen as a classical routing problem where the order of sending the packages has to be determined. Seeing that the capacity of the grid is limited, it still needs to be guaranteed that the packages arrive securely and in an efficient manner.
The use of P2P DET brings some issues concerning security and privacy. End users in the grid need to communicate with each other to ensure the success of the trading process. In recent research they enabled peers in a smart grid to anonymously negotiate energy prices and securely perform trading transactions [3]. However, this issue is not always addressed.

3 Methodology of the Comparative Review

In this study we want to give a better understanding of P2P DET challenges and for this reason give a comparison of recent studies on this topic. Our main focus will be on energy storing used for the trading, the grid structure, how supply and demand are balanced/matched, power routing and an examination how the paper determines the success of the proposed findings of their research.

3.1 Energy Storage

The hardware requirement for each grid can differ for the model or architecture someone chooses. The details could make a difference in the usability of the grid and the difficulty of implementing it. Energy storage capacities in a grid architecture have the big advantage of stabilizing the energy flow in the grid. While battery systems are not yet scalable for grid-scale application, they are on a promising path to become widely available soon [2]. For the purpose of the comparison, each proposed trading structure is reviewed if energy storage (ES) is included.

3.2 Grid structure

A grid used for P2P DET system can have a wide range of components such as conventional power plants, distributed energy resources, transformers, power routers, smart meters, etc. Furthermore, it can have many different stakeholders like the traditional consumers, producers like retailers, and prosumers equipped with solar panels and wind turbines in their household. The grid size can reach from one single household, to neighborhoods to whole states and nations, and it can be integrated into the conventional utility grid, or completely isolated and self-sustained. This paper will investigate which kind of grid structure each of the four approaches suggests or presupposes for their trading system.

3.3 Load Balancing

As mentioned earlier, smart grids offer new opportunities but also challenges to the task of keeping supply and demand at balance. This paper examines how each proposed trading system tackles this task and compare them to each other.

3.4 Power Routing

Power routing is considered to be a necessity for each P2P trading system. We check for each study what infrastructure they use to enable bidirectional energy flow and how energy is efficiently transported from generation side to the demand side.

4 Selection of P2P DET Solutions

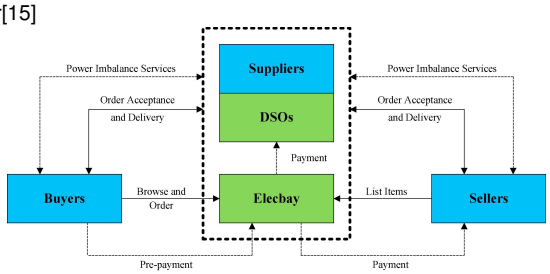
This section gives a summary of the four P2P energy trading architectures that we selected for a detailed comparison.

4.1 P2P Energy Trading in a Microgrid

The goal of Long et al. [15] was to develop a platform for the "business layer", as they call it, of P2P DET. This layer is concerned with organizing how power is traded between prosumers and is considered independent from the "power grid layer", the "ICT layer" and the "control layer" which deal with the physical implementation of energy transmission, routing, and grid synchronization respectively. The software they propose [15] is called "Elecbay" and performs P2P energy trading in three phases. In the beginning, the "bidding" phase; each prosumer who wants to sell energy submits their offers, which consists of 30 minute packages of power, and with it the desired price. In this phase, prosumers with energy demand can choose from these offers and place orders. Finally, the system decides which transaction will be accepted. How users in the grid interact with the platform can be seen in Figure 1.
In the following phase the energy exchange takes place, in which both promised production and consumption levels are measured precisely by smart metering devices. This is done to ensure that in the last phase, the settlement, energy consumers can be billed for the correct amount and any deviance from the stated energy levels can be financially punished.
The main goals of the platform proposed in paper [15] is to offer prosumers the opportunity to trade energy in such a way that the consumed energy comes as much as possible from local sources and not from the external utility grid, and secondly, that the flexible demand is scheduled in a way that maximizes their payoff. Flexible demand includes all those appliances that can be used relatively time independent such as washing machines, dishwashers and water heaters.

Prosumers thus have an advantage when they schedule their flexible demand during a time when they themselves produce a lot of energy, or when energy on Elecbay is cheap.
Simulation results showed, that if the RES are sufficiently diverse, energy exchange between the utility grid and the microgrid is reduced by 42.49% and the peak load of the microgrid was reduced by 17.6%.

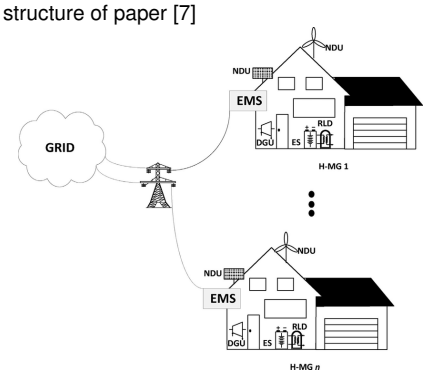
Fig. 1. Interactions of grid participants during P2P energy trading in paper[15]



4.2 Retail Electricity Market and Home Microgrids

The platform that is presented by Marzband et al. [7] presupposes a set of prosumers with RES, energy storage capacities, and a smart energy management system that connects the house to the utility grid, shown in Figure 2. Through the grid, each prosumer is also connected to a "market operator" and multiple energy retailers.
The market operator is the core of the platform as it receives from each participant in the market - both prosumers and retailers - supply and demand bids for the following day. From this, it computes the optimal price and the optimal schedules of the prosumers' home systems and the retailers' energy production systems. The bids are essentially predictions about the amount of energy that will be produced or consumed, together with the desired price, while the schedules prescribe at what time flexible demand should be allocated.
The market system is set up in such a way that firstly, prosumer produced energy is sold as much as possible to other prosumers and any excess power is sold to the retailers, who resell it when demand exceeds supply. The authors claim that this market structure maximizes the use of RES through the optimized storage and demand response schedules, that are found by the market operator and reduces the energy price because of more competition.

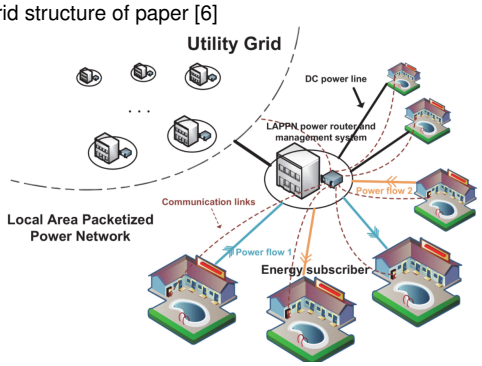
Fig. 2. Grid structure of paper [7]



4.3 Power Dispatching for Local Packetized Power Networks

J. Ma et al. [6] propose a protocol that achieves both handling Response Demand and Power Routing by separating the energy transaction process into three parts.
The first part they call "subscriber matching" in which for each "subscriber", i.e. a prosumer, who has a certain amount of surplus energy to sell a corresponding other subscriber is found, who has the demand for that amount of energy.
In the "transmission scheduling" step it is determined when each package is sent over each of the limited power channels of the power router. The third step is then sending the energy packages in the "power packet transmission" after confirmation from each prosumer was obtained. This protocol was developed for a local network consisting of a small number of neighbors, each with their own IP address, AC/DC converters and battery systems, connected to one power router that also has a connection to the conventional utility grid (Figure 3). The power router possesses multiple power channels through which only one package at a time is sent, so that it can achieve peer-to-peer trading for multiple pairs at once. It is connected to the utility grid to that it can still supply energy to those prosumers that did not find a partner in step 1.

Fig. 3. Grid structure of paper [6]



4.4 Energy Routing in a Smart Grid Network

The algorithm proposed by J. Hong et al. [5] aims at finding the optimal energy price for each prosumer, so that their profits are maximized, while at the same time finding the optimal power transmission path. To do so, they present their idea of a "control center" that receives information from prosumers about how much electricity they intend to buy or sell together with their desired prices. The control center then determines the optimal price through a stock exchange pricing scheme and finds the optimal transmission path by using the Hungarian algorithm, a popular solution to the transportation problem, which is of the same structure as the power routing problem according to paper [5].
The underlying grid structure that the authors presuppose for their algorithm is a smart microgrid that connects houses in one neighborhood (Figure 4). Each house must be equipped with RES, energy storage capacities and a power router. It is neither specified how the control center is connected to the grid, nor is power conversion mentioned.

Fig. 4. Microgrid structure in paper [5]

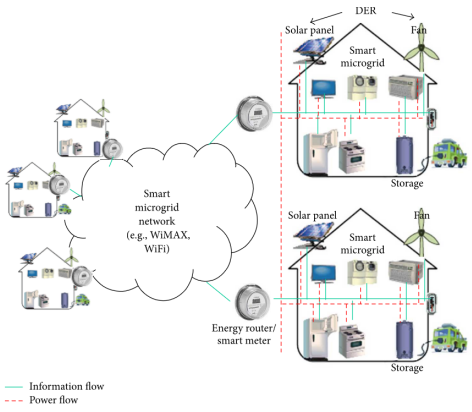


Table 1. Comparison of P2P DET solutions

Paper reference #	[15]	[7]	[6]	[5]
What did they measure to determine that their approach was successful?	Using the p2p platform with various DER's leads to a peak load reduction of 17.6% and a reduction of energy exchange with the utility grid of 42.49%; able to improve the local balance of energy generation and consumption.	In the case with most competition, total generated energy increased by over 300% and total consumed energy increased by over 200%, while the energy price was lower. More flexible demand could be satisfied.	The transaction matching process was compared with a random base case and it was significantly better. It was also examined whether prosumers with a price advantage were scheduled earlier, which was the case, so the algorithm was deemed fair.	The authors tested the algorithm's ability to keep the balance between supply and demand in 3 different use cases. The algorithm was successful in each case.
How does the grid structure look like?	Undefined, general structure that can be applied on any scale to the grid	Individual prosumers are connected to the grid	Local Networks that are connected to a conventional grid	Isolated smart grid
Who are the participants in the grid?	Prosumers, utility companies, Elecbay	Prosumers, utility companies, market operator	Prosumers, utility companies	Prosumers
Is ES integrated in the grid	no	yes	yes	yes
Power routers?	yes	no	yes	yes
Is energy conversion addressed?	no	no	Each household has converters	no
Load Matching technique	Demand Response	Demand Response	Other	Other
Privacy/security	not mentioned	address one more secure optimal scheduling option	not mentioned	not mentioned

5 Research Results

The authors of each paper tested their approaches in simulation experiments. The results of these experiments are summarized in Table 1. Nevertheless, the experimental results are not trivially comparable because of the different research focuses of the papers. While C. Zhang et al. [15] and M. Marzband [7] propose actual trading platforms, papers J. Ma et al. [6] and J. Hong et al. [5] propose protocols that are less concerned with the business side of DET, and more concerned with certain technical details of it. Therefore, in each study the algorithm's performance was tested in regards to different aspects, which makes it not possible to compare the effectiveness of the algorithm's with each other, without running further simulations on each of them. Nonetheless, comparing the approaches in regards to more qualitative aspects that are also shown in Table 1 and the results of this comparison are presented in this section.

The P2P DET approaches in papers [7], [6] and [5] require energy storage capacities to function successfully. They are needed to decouple the time of energy creation from the time of consumption, so that energy can be sold on a market like any other commodity [2]. However, C. Zhang et al. [15] did not include energy storage in their case study and only proposes a system that is able to keep the balance of supply and demand solely through demand response. This makes it possible to implement it with current technology.

While papers [15], [7], and [6] present architectures in which the prosumers are still connected to traditional power companies through the utility grid, Wang et al. [5] present a microgrid that is completely independent from the conventional utility grid. Here the demand has to be fully covered by other prosumers from inside the grid.

What they all share, is a centralized institution that receives information from all participants about how much energy they want to sell/buy, usually also when they need it, and what price they desire. Even in paper [15] where prosumers can browse other prosumers' offer and choose what to buy themselves, this institution is ultimately in control of who gets to send energy to whom, and for what price.

In papers [6] and [5] it is especially important that this is decided centrally, because in this process, it is secured that supply always matches demand. They have to rely on these algorithms, since neither of the two approaches provides any mechanism for demand response.

Regarding the infrastructure of power routing, only J. Hong et al. [7] do not explicitly mention energy routers which are responsible for handling the transportation of electricity. However, the authors do presuppose a home energy management system which is "able to send/receive signals to/from a market operator" [7] and has additional functionality, as it must predict load demand and energy generation for the next day.

Papers [6] and [5] are the only studies that are concerned with the optimality of the route that the electricity takes through the grid. While paper [6] is more concerned with the fair and optimal scheduling of data packages through certain choke points of the grid, paper [5] finds the optimal path on which the power can be sent.

Only M. Marzband et al. [7] address the issue of privacy in security of their platform, as they discuss that there are two ways of computing the optimal schedules of the prosumers; either it happens centrally in the market operator, or each home energy management system does it locally, which is more secure, but poses higher upfront costs to the prosumers, which discourages participation in the system.

6 Discussion

This contribution's aim was to give an introduction to the opportunities and challenges of P2P DET in smart grids and to give an overview of how possible implementations of it could look like.

It is interesting to note that most of the papers included ES in their grid architecture. These findings suggest that ES will be an important part of the development of smart grids and P2P DET systems. As a possible future instantiation of ES in the grid we propose distributed energy storage systems such as the usage of electric vehicles that are charged during times of excess energy production or home batteries such as the Tesla Powerwall [11].

With a growing amount of data and machine learning solutions in context of smart grids [4, 13], it is fitting to see that two of the four selected approaches rely heavily on smart systems that are able to give day-ahead predictions of energy demand and production levels. These systems could be a key component of effective demand response systems in the future.

Further, we will discuss the limitations of this review. First of all, it is clear that this was not an exhaustive survey of the literature on peer-to-peer energy trading in microgrids. Our selection of studies was based on our impression of how notable each of them are in this field of research. Further surveys with larger scopes could be conducted to get a more representative image of the state of research on this topic.

Another limitation was the fact that we could not make any comparison between the efficacies of the approaches, because they all focused on different aspects of their simulation results, even though they all implemented very similar systems. In order to compare their effectiveness at e.g. reducing the maximum power load of the grid, or at reducing the power loss of energy transportation through more optimal routing, all platforms would have to be simulated under the same conditions. Such a simulation could be the focus of further research.

This paper focuses more on the technical side of P2P DET. In future work, we could extend the review by offering a multi sided perspective of P2P DET by also looking at the policy, laws, ethics and economics that are affected implementing the P2P energy trading. Aforementioned paper [7] is the only research that gave an more secure option to calculate the optimal optimal schedules of the prosumers. Besides, the other papers do not even address privacy and security challenges of P2P DET nor included it into their given architecture. Further research should be undertaken to investigate this issue in P2P DET.

7 Conclusion

Proposed is a review on existing research related to P2P DET. This paper reviews the possibilities of integrating P2P energy trading solutions from a technical perspective. For different papers we looked at their focus including energy storage, grid structure, load balancing and power routing. Some of the papers share similarities. However, they do have different focus for their research. Provided is a comparison table of these findings. Energy storage was used a lot in the grid to be a very promising technique. All approaches take care of the load matching challenge, while the power routing challenge is only addressed in papers [6] and [5].

Instead of only investigating the technical sides, further research could explore the political, economical and ethical implications of P2P DET and its security issues.

Acknowledgements

The authors wish to thank Ang Sha for providing the research topic

References

[1] J. Abdella and K. Shuaib. Peer to peer distributed energy trading in smart grids: A survey. *Energies*, 11:1560, 06 2018.

[2] R. Abe, H. Taoka, and D. McQuilkin. Digital grid: Communicative electrical grids of the future. *2010 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT Europe)*, pages 1–8, 2010.

[3] N. Aitzhan and D. Svetinovic. Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams. *IEEE Transactions on Dependable and Secure Computing*, PP:1–1, 10 2016.

[4] D. Alahakoon and X. Yu. Smart electricity meter data intelligence for future energy systems: A survey. *IEEE Transactions on Industrial Informatics*, 12(1):425–436, Feb 2016.

[5] J. Hong and M. Kim. Game-theory-based approach for energy routing in a smart grid network. *Journal of Computer Networks and Communications*, 2016:1–8, 01 2016.

[6] J. Ma, L. Song, and Y. Li. Optimal power dispatching for local area packetized power network. *IEEE Transactions on Smart Grid*, 9(5):4765–4776, Sep. 2018.

[7] M. Marzband, M. Javadi, S. A. Pourmousavi, and G. Lightbody. An advanced retail electricity market for active distribution systems and home microgrid interoperability based on game theory. *Electric Power Systems Research*, 157:187 – 199, 2018.

[8] N. Saputro, K. Akkaya, and S. Uludag. A survey of routing protocols for smart grid communications. *Computer Networks*, 56:27422771, 07 2012.

[9] T. Takuno, M. Koyama, and T. Hikihara. In-home power distribution systems by circuit switching and power packet dispatching. In *Proceedings of the 2010 First IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 427–430, October 2010.

[10] K. Tashiro, R. Takahashi, and T. Hikihara. Feasibility of power packet dispatching at in-home dc distribution network. In *Proceedings of the 2012 IEEE Third International Conference on Smart Grid Communications (SmartGridComm)*, page 401405, November 2012.

[11] C. N. Truong, M. Naumann, R. C. Karl, M. Mller, A. Jossen, and H. C. Hesse. Economics of residential photovoltaic battery systems in germany: The case of teslas powerwall. *Batteries*, 2(2), 2016.

[12] J. S. Vardakas, N. Zorba, and C. V. Verikoukis. A survey on demand response programs in smart grids: Pricing methods and optimization algorithms. *IEEE Communications Surveys Tutorials*, 17(1):152–178, Firstquarter 2015.

[13] H. Xu, H. Huang, R. S. Khalid, and H. Yu. Distributed machine learning based smart-grid energy management with occupant cognition. In *2016 IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 491–496, Nov 2016.

[14] C. Zhang, J. Wu, C. Long, and M. Cheng. Review of existing peer-to-peer energy trading projects. *Energy Procedia*, 105:2563 – 2568, 2017. 8th International Conference on Applied Energy, ICAE2016, 8-11 October 2016, Beijing, China.

[15] C. Zhang, J. Wu, Y. Zhou, M. Cheng, and C. Long. Peer-to-peer energy trading in a microgrid. *Applied Energy*, 220:1 – 12, 2018.

Ensuring correctness of communication-centric software systems

Rick de Jonge, Mathijs de Jager

Abstract—In the past, various approaches for assurance of correctness of software systems have been proposed. Examples include Session Types and Conversation Calculus. In this paper we first lay the basis with a global overview of the π-calculus both calculi are based upon, followed by summarizing these two approaches. This summary describes the core philosophy the calculi were based on and how the language can be used in practice. This includes an example of a word storing service with which the differences between the low-level decisions of the Session Types and the higher-level design principles of Conversation Calculus are made clearer.

After this overview, we compare the differences of approach the calculi took to represent communication-centric services. We also compare their practical use, with the example we use throughout the paper as a base. We conclude that Session Types are more programmatic, using lower-level structures like data-typing to represent smaller systems, while the Conversation Calculus is a good representation of a higher level design with its notion of conversations and the flexibility these conversations have.

Index Terms—Service Oriented Computing, Type Systems, Process Calculi, Program Correctness, Distributed Software

1 Introduction

Software systems can be proven to be working as they were intended, but this is a difficult task for communication-centric software systems such as a client-server setup. Such software systems often involve concurrency to increase the speed that a single session can be processed with, or to accept multiple communications at the same time. Compilers verify whether the code is correct to the extent that it can be built. Additionally, unit testing can evaluate correctness via input and output of sub-parts of the software system. However, it generally cannot tell if the steps to achieve the output within the system went according to the intended specifications.

Essentially, to ensure the correctness of communication-centric systems, the communication that occurs in the system must adhere to a protocol definition. A multitude of ways of doing so can be found in the current literature. Conversation Calculus [13] and Session Types [7], both extensions of the π-calculus, have been used to represent the flow of such a system. Conversation Calculus mostly focuses on letting whole processes communicate more freely with eachother, while research using Session Types has implemented a way of ensuring correctness closer to widely used programming languages. These approaches have to be able to express different ways of communication.

We want to explore the differences between these two approaches, with the main focus to find out how practical these approaches are. We will work towards our research question: *What is the difference between two approaches to ensure the correctness of communication-centric software systems, Session Types and Conversation Calculus, and how can they be integrated into building software?*

In this paper, we will first describe the π-calculus followed by two extensions to it, Conversation Calculus and Session Types. For each of the latter two, we will explore why they were created, how they express communication systems, and how the approach has been used in applications. We will compare these two approaches in the section afterwards, exploring which systems the approaches are best used to inspect.

- *Mathijs de Jager, Msc. Computing Science student at the University of Groningen, Email: m.de.jager.8@student.rug.nl*
- *Rick de Jonge, Msc. Computing Science student at the University of Groningen, Email: g.d.de.jonge@student.rug.nl*

2 π-calculus

In order to grasp the concepts behind the approaches which we will consider later on, we need to briefly point out the basics of π-calculus.

π-calculus is a type of process calculus. Process calculi are formal languages typically used to model concurrent systems. Over the years, various variants of process calculi have been developed. The aim remains the same across the approaches. However, they do vary in expressive power. If a process calculus is more expressive, then it can model the concurrent system more accurately.

π-calculus is more specific than many other process calculi in the sense that it allows the topology of the model to change. It does so by passing channel links through channels, such that endpoints of channels can change. By allowing the calculus to define such properties, its expressive power increases.

Construct	Notation
Concurrency	$P \mid Q$
Input-prefixed process	$c(x).P$
Output-prefixed process	$\bar{c}\langle v \rangle.P$
Replication	$!P$
Create constant	$(vx)P$
Nil process	0

Table 1. Summary of constructs in π-calculus

An overview of the basic constructs in π-calculus and their corresponding notations can be found in Table 1. Note that the approaches discussed in this paper build upon these constructs. Probably the easiest way to grasp the basic concept is to think of the model as channels connected to each other, each expressing their expected input and output. The channels function as communication links.

3 Session Types

In this section we will discuss the first of the two methods, the Session Types. Session Types were originally introduced by Gay [8] as an extension to π-calculus. The extensions effectively consist of either adding a labelled choice or choosing a labelled choice. Despite being expressive, π-calculus in itself has no notion of data types. Without typing, it is harder to explicitly model a communication protocol.

As an example, there is no notation to specify the input type for a channel. The input of a channel could be anything. This means that without the possibility of restricting a channel to a certain type of interaction, we are missing some expressiveness.

A system represented using the Session Types [7] syntax can be divided into multiple sessions. Each session represents a single user performing a series of actions. If one of these actions is a choice, we have multiple paths to follow. These paths can be represented in the same session as well. This representation is indented, making it easy to follow the different choices that can be made. These session representations, consisting of input and output, need to have a counterpart in another part of the system for it to be a correct system. The counterpart can be constructed by so-called dual functions. They allow us to check the correctness of this communication-centric software system.

Session Types try to extend correctness techniques to accommodate for new programming methods. In doing this, it tries to remain close to programming styles like object-oriented programming and functional programming for easier implementation in those languages. Session Types have been extended in various ways, including adding multiple parties to sessions [2], and including a way to place constraints when initiating a session [6]. These new extensions allow the language to represent a larger set of systems with different and more specific communication design models.

Expressing Session Types in different languages is possible, but some extra care is needed in order to do so. π-calculus inherently communicates through channels, which might not be the case for other languages. Various constructs are added for functional and object-oriented languages to be able to express the same Session Types. Vasconcelos [12] has augmented Session Types with constructs that were found needed for both a functional and an object-oriented language.

Session Types have been implemented into a couple of real-world programming languages, including *Haskell* [10] and *Scala* [11]. Implementing Session Types into a language proves they can be used for more than strictly a theoretical tool for ensuring correctness. Haskell is a functional language, while Scala is object-oriented with many functional features. The difference in programming paradigms in the two examples shows that Session Types can be used for both types of languages.

3.1 Features and notation

As mentioned in the previous section, Session Types build upon the notation introduced by π-calculus. There is some additional syntax that describe the features introduced by Session Types, which we will shortly describe here.

In order to denote whether something is input, a question mark (**?**) is used. For output, an exclamation mark (**!**) is written down.

Branching is indicated with an ampersand (&) and denotes multiple paths that can be taken. Which path is followed depends on the input to the system. Often, this choice is made by input from the other side of the session. This choice is denoted by $\oplus$, followed by $n \geq 1$ amount of sub-paths.

Termination of a session is indicated by **end**. After such a statement, one can expect the session to be closed.

There are a couple of predefined basic types. Some examples include *Id*, *String* and *Int*. The meaning is easily inferred from the name. The exact definition of a certain type does not matter, as long as the participating components have the same understanding of the type.

3.2 Proving correctness

The example in Listing 1 shows a simple definition of a possible interaction described using Session Types. Effectively, it denotes a word saving service: it can save or retrieve a word to or from a cell. The upper definition resembles a client sending a string to the server. The lower part defines the server part of the definition. The expected types are denoted by a type name (such as *String*). While this example is very simple compared to real communications within a software system, it suffices as an illustration of differences in syntax between the approaches.

```
!String. &{ success: (+){ save: !String
                        | retrieve: &String
                        | stop: end
                        }
          | failure: end
}
```

Listing 1. An example with Session Types describing a simple *String* storing service, with options to store a string, retrieve it or stop the service.

```
?String. (+){ success: &{ save: &String
                        | retrieve: !String
                        | stop: end
                        }
            | failure: end
}
```

Listing 2. An example with Session Types describing a user that can interact with the simple string storing mechanism from Listing 1

A concrete example of a session represented by the Session Type definition in Listing 1 can be found in the diagram of Figure 1. Both possible paths that can be enacted (the trivial *stop* command excluded) are shown there.

4 CONVERSATION-CALCULUS

We will now discuss another method of proving correctness, the Conversation Calculus. Conversation Calculus, another adaptation of the π-calculus, as presented in [13] [5] [4], bases itself around the concepts of instances in a conversation context. These conversations represent a communication channel between processes in which interactions between parties can be proven to be complements. Ensuring the correctness of the system as a whole by proving correctness of its parts. The authors say they intended the Conversation Calculus to be clear, simple and minimalistic, while still being very expressive.

4.1 Fundamental features of Conversation Calculus

Conversation Calculus defines every process as a part of a conversation. Conversation contexts can be created and joined by these processes. In such a context, an instance can define functions consisting of input and output messages, enhanced with context awareness, parallelism, variable restriction. Conversation Calculus also has a definition for exceptions at its core.

Conversation calculus focuses on a few fundamental features of a communication-centric system, namely distribution, process delegation, communication and context sensitiveness, and loose coupling.

4.1.1 Distribution

Distribution is a core aspect of a communication-centric system. Such a system needs to make sure that activities are divisible. Dividing processes in small functions that each have their own purpose is a core design principle in many programming tasks. Since distribution of services is close to the core of how programmers design programs, this allows the Conversation Calculus to be implemented more easily into new designs. Furthermore, each process needs to be responsible for allocated activities and resources need to be properly divided among these processes.

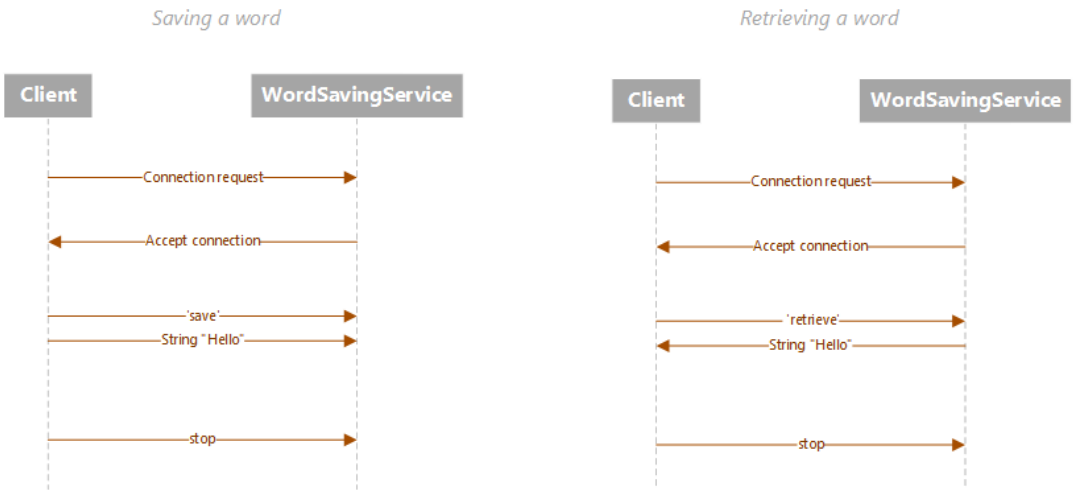


Fig. 1. Retrieving and saving a word from the service. The client can save a *String* on the service (left) or retrieve the stored word from it (right).

4.1.2 Process delegation

Process delegation is used in a system to divide activities. This will be done by sending complete methods over a communication channel and additionally a way to invoke these methods. This principle further enforces the system to use a master-slave design. There is a master that distributes the processes among the slaves and invokes these processes with data. In the language, these processes can be defined with the **def** keyword and instantiated with the keyword **instance** or **new**. After creation, they can be joined by other processes with the keyword **join**.

4.1.3 Communication and context sensitiveness

Communication and context sensitiveness allows processes in a system to join certain groups in which information can be shared. Information will not be relevant outside these contexts and information outside these contexts should not be relevant to this context. Only the processes inside these contexts are able to interact with each other using this information. These contexts are at the core of Conversation Calculus. Each process is part of such a context, called a conversation. In the language an *initiator* is denoted by a left-pointing black triangle (◄) and a *responder* by a right-pointing black triangle (►), meaning that processes are located at an initiator endpoint and responder endpoint respectively. The communication itself is denoted by the keyword **in** or a question mark (**?**) and the keyword **out** or an exclamation mark (**!**) for input and output respectively.

4.1.4 Loose coupling

Loose coupling ensures that little information is shared between processes. Similar to contexts, which ensure closed communication, loose coupling ensures that a process is responsible for itself and its task. This process should need as little outside interaction as possible. The difference between loose coupling and sensitiveness discussed in the previous paragraph is that here, the information should not be needed outside the conversation while before the information should be protected from outside sources.

4.1.5 Exception handling

A minor feature not listed as main feature but still prominently featured is exception handling. This is a more practical view towards use of Conversation Calculus to use during design of a system. Exception interactions are denoted in the language with the keywords **try**... **catch** to express the handling of an exception and the **throw** keyword to create one. This allows systems to break from the defined structure and still be able to function when unexpected problems occur.

4.2 Representing systems in Conversation Calculus

Using Conversation Calculus, we can represent a variety of communication-centric software systems. With the basic definitions, some of them listed in the previous section, we can define processes and conversation for them to join. We can send and receive data from and to endpoints and ensure the correctness of such a basic communication. A system in Conversation Calculus is divided in processes. Each of these processes will then consist of branched paths of input and output functions, sending or receiving data in any channels they have created or joined. This has native support for multi-process channels and parallel processes, allowing for multiple processes to use a single service.

Given all these concepts in the previous paragraphs, here is another example of the word saving cell, but now in the Conversation Calculus:

```
new WordSavingService . Cell [
  !(
    save?(x)
    +
    retrieve?().reply!(x)
  )
  | stop?()
]
```

Listing 3. Conversation Calculus describing a simple *String* storing service, with options to store a string, retrieve it or stop the service.

In this example, the *new* keyword lets us define a conversation that can be joined by our newly created *Cell* function. The + key indicates a choice. Here, another service in the conversation is expected to send a request to *save* or *retrieve* a word. In the case the *save* function is called with a new word, this word is saved. In the case of a *retrieve*, the stored word is send to the conversation. Since this is a running service that can accept multiple *save* and *retrieve* commands, we want to be able to stop the service when needed. When this is the case, a stop signal can be send to stop the service, an action that uses the concurrency used from π-calculus.

4.3 Proving correctness

When a conversation and the services within are defined, they will still need to be proven.

If we want to join our existing conversation with a service that chooses to either save or retrieve a word from the *Cell*, we can do that using the definition in Listing 4.

```
WordSavingService ◄ [
  !(
    save!(x)
    +
    retrieve!().reply?(x)
  )
  | stop!()
]
```

Listing 4. Conversation Calculus describing a user that can interact with the simple string storing mechanism from Listing 3

In this simple case, the service is joined, not created and the input and output symbols have been switched.

4.4 Integration of the Conversation Calculus

The Conversation Calculus was introduced with and created for the Sensoria Project [1], a project to fully integrate theory in a pragmatic software engineering approach, this in a more understandable way. After a few failed ideas, it was started around the idea of conversations, with a comprehensive and therefore basic communication foundation. The biggest improvement over other types of process calculi is the addition of exceptions, a field left mostly unexplored but very practical. The perspective Conversation Calculus takes with its notion of conversations also resembles an existing implementation slightly, Boxed Ambients [3], which is better known in the literature.

5 Discussion

In this section we will compare the approaches to identify their specific solutions to certain aspects. Moreover, we try to determine which approach is a good fit for ensuring correctness in a communication-centric system.

5.1 Approach to the problem

Both Session Types and Conversation Calculus approach designing communication-centric systems by extending the working π-calculus with types already present in programming languages. Given the narrow field these calculi are developed for, more specific constructs can be added to the calculi. Session Types focus on adding datatypes to ensure correctness in the details, allowing designers to represent specific static communications between processes. This gives rise to easy translations and thus implementations into the modern functional and object oriented paradigms of programming. On the contrary, the Conversation Calculus approaches the problem from a perspective that is much more broad. It allows for a more flexible way of defining communication by allowing processes to join and leave conversation contexts. However it appears the approach may be grounded too much in the project it was built for.

5.2 Practical application

We can look at the usage of the previously introduced calculi in the literature to gain an understanding of the practical use of these techniques. Without a practical implementation, it would be up to the user to manually check correctness. While there are few implementations besides the Sensoria Project [1] for the Conversation Calculus, various papers describing Session Types implemented in programming languages with real-world adoption such as *Haskell* [10] and *Scala* [11] have been published. This fact suggests Session Types has seen more real-world usage than Conversation Calculus. In spite of that, this does not necessarily tell us much about whether Session Types is a better fit for ensuring correctness in communication-centric systems.

5.3 Comparing the Word Saving Cell

Earlier, we gave the same example in both the Session Types and the Conversation Calculus syntax. We will now illustrate the points made earlier in this discussion using this example.

The absence of any context is notable in the Session Types. While a session channel can be specified, the ability to leave it out enhances the idea that the Session Types are more focused on the actual communication between multiple services.

Furthermore, a fundamental change is the typing of variables. The *String* type is prominent throughout the Session Types example, while the Conversation Calculus only allows the naming of such a variable, like the variable x in the example. Both these approaches are a useful extension of the π-calculus. While variable naming can be used widely, typing variables is very useful only for programming and thus low-level design.

The final notable difference is the specific acceptance of the communication in the Session Types example. While this could be a part of the conversation keywords in the Conversation Calculus, this is not translated as easily into actual programming software.

6 Conclusion

In the introduction, we posed a question:

What is the difference between two approaches to ensure the correctness of communication-centric software systems, Session Types and Conversation Calculus, and how can they be integrated into building software?

We saw that the two approaches differed in their core goals, Session Types aimed to implement additional features to an existing formula which got successful practical implementations. Conversation Calculus changed the core of the existing formula to create a niche complete implementation that resulted in a completed project, but not much attention from developers to apply the approach to software systems.

The main difference between the Session Types and the Conversation Calculus approach seems to be that Session Types turns out to be more useful for low-level programming design choices, while Conversation Calculus can easier be used to create a higher level design. The inclusion of data types and a bigger focus on static sessions in the Session Types approach oppose the dynamic interaction with conversations of the Conversation Calculus. The quick translation from calculus to code is likely why Session Types are currently being used in programming languages, while the Conversation Calculus is too general and perhaps too big to be of practical use.

These conclusions are not completely in favour of using Session Types over Conversation Calculus, however. One argument against this conclusion is the extension of exception handing that is present in the Conversation Calculus and not with Session Types. Exception handing would be more useful to combat incorrect low-level implementations, for instance when passing an illegal argument. In the end, there is no definite best approach to use in general, each of the two researched approaches can help designing a project.

7 Future research

This paper only considers Session Types and Conversation Calculus. Extending this comparison to the other approaches to this problem would be a logical next step. Possibly, behavioral types [9] and its notions can be included in the comparison. For example, behavioural contracts might show some parallels to the approaches discussed in this paper.

Acknowledgements. We thank Jorge A. Prez for reviewing and providing feedback on this paper. We also want to thank our colleagues Lars Doorenbos and Hayo Ottens for their feedback and insights. Finally, we thank our professors at the University of Groningen for organizing the reviews and publications.

REFERENCES

[1] P. Baldan, M. Bartoletti, and B. Baudry. Ip sensoria project, 2005-2010.

[2] L. Bettini, M. Coppo, L. D'Antoni, M. De Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In F. van Breugel and M. Chechik, editors, *CONCUR 2008 - Concurrency Theory*, pages 418–433, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[3] M. Bugliesi, G. Castagna, and S. Crafa. Access control for mobile agents: The calculus of boxed ambients. *ACM Transactions on Programming Languages and Systems*, 26:57–124, 01 2004.

[4] L. Caires and H. T. Vieira. Conversation types. In G. Castagna, editor, *Programming Languages and Systems*, pages 285–300, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[5] L. Caires and H. T. Vieira. Analysis of service oriented software systems with the conversation calculus. In *Proceedings of the 7th International Conference on Formal Aspects of Component Software*, FACS'10, pages 6–33, Berlin, Heidelberg, 2012. Springer-Verlag.

[6] M. Coppo and M. Dezani-Ciancaglini. Structured communications with concurrent constraints. In C. Kaklamanis and F. Nielson, editors, *Trustworthy Global Computing*, pages 104–125, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[7] M. Dezani-Ciancaglini and U. de'Liguoro. Sessions and session types: An overview. In C. Laneve and J. Su, editors, *Web Services and Formal Methods*, pages 1–28, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[8] S. Gay. Types and subtypes for client-server interactions. 06 2000.

[9] H. Hyttel, E. Tuosto, H. Vieira, G. Zavattaro, I. Lanese, V. Vasconcelos, L. Caires, M. Carbone, P.-M. Denilou, D. Mostrous, L. Padovani, and A. Ravara. Foundations of session types and behavioural contracts. *ACM Computing Surveys*, 49:1–36, 04 2016.

[10] M. Neubauer and P. Thiemann. An implementation of session types. 05 2004.

[11] A. Scalas and N. Yoshida. Lightweight session programming in scala (artifact). 01 2016.

[12] V. Vasconcelos. Sessions, from types to programming languages. *Bulletin of the European Association for Theoretical Computer Science EATCS*, 103, 01 2011.

[13] H. T. Vieira, L. Caires, and J. C. Seco. The conversation calculus: A model of service-oriented computation. In S. Drossopoulou, editor, *Programming Languages and Systems*, pages 269–283, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

A Comparative Study of Random Forest and Its Probabilistic Variant

Zahra Putri Fitrianti and Codrut-Andrei Diaconu

Abstract— Machine Learning algorithms have become an important tool in data analysis. They can be used in various tasks, classification being one of the most common where we aim to predict the labels/classes of the data points using their attributes. An example of such algorithm is Random Forest (RF) which is an ensemble method consisting of multiple trees, such that each tree uses a random subset of features and a random subset of the training data points. However, some algorithms do not necessarily perform well when the data has noise, e.g. errors introduced by measurement tools, missing values etc. In order to take these uncertainties into account, the Probabilistic Random Forest (PRF) was implemented in such a way that it treats the features and labels as probability density functions. This makes the model more robust compared to the standard RF where the deterministic values are used. We compared the standard RF and PRF by performing some tests on benchmark data sets and analyze the results in terms of both performance (e.g. accuracy, F1-score, ROC AUC) and computation time. We observed that PRF has a better performance especially when the fraction of missing data is high but it comes with a significant increase in computation time.

Index Terms—Classification, Ensemble, Random Forest, Probabilistic Random Forest

1 Introduction

Machine Learning algorithms have become an important tool in data analysis. They can be used in various tasks, for example in unsupervised learning, supervised learning, and many more. For the scope of this paper, we will only consider supervised learning as it is related to the random forest algorithm. Supervised learning is a method that takes a set of labelled examples as input and tries to predict the output as a function of the input. In other words, supervised learning maps the labelled input into an output [2]. There are many examples of supervised learning but a very popular one is classification where we assign classes/labels to data points using their features.

Typically, in classification, a training-testing workflow is used which means that first, a model (i.e. the classifier) is trained on a portion (e.g. 80%) of the available labelled data and the remaining part is used to test the model. This testing phase plays a very important role since it provides a means to assess the generalization performance of the model when tested with an unseen example. Some performance measures can be used for evaluating the quality of the predictions, such as accuracy score, F1-score, precision, recall, Mathew Correlation Coefficient (MCC), Area under ROC curve, and many more. The choice depends on the type of the data and on what aspects are more relevant to the problem at hand. For instance, in medical applications, usually the focus is on minimizing the risk of false negatives (i.e. not detecting a disease when it is present) and not on the overall accuracy which is usually high in this situation since most of the people are not sick. In other situations, the aim is just to maximize the accuracy (e.g. in optical character recognition tasks). Pertaining to this paper, we will use only three performance metrics namely accuracy, F1-score and the Area under ROC curve.

There have been several approaches that we can take in order to improve the performance of an algorithm, one of which is combining multiple learners or often referred to as ensemble method. This can be done in many different ways: apply cross validation to the data set such that each base-learner uses different training sets, use different algorithms to train each base-learner, or train all with the same algorithm but use different parameter settings [3]. One main advantage of creating an ensemble is improving prediction performance. This is due to the fact that usually, an ensemble classifier performs classification by means of majority voting so the final prediction outcome is given by the majority of the prediction yielded by the base learners. Subsequently, this will improve the prediction ability of the ensemble classifier because, it could be the case that different base-learners have poor performance on a novel input but good performance when they are combined.

One particular example of ensemble methods is Random Forest (RF) where several decision trees are combined such that each tree depends on the random features and/or subsamples that are independent and identically distributed. Then, to yield a prediction of an example, it is propagated through the tree by evaluating the features until it reaches the leaf node. The resulting label would then be the class which has the highest probability in the leaf node [5]. It is important to note that RF treats the features and labels as deterministic values. Hence, it might not necessarily perform well when the data has noise, e.g. errors introduced by measurement tools, missing values etc. In order to take these uncertainties into account, the Probabilistic Random Forest (PRF) was implemented in such a way that it treats the features and labels as probability density functions [14]. This makes the model more robust compared to the standard RF where the deterministic values are used.

In [14], the authors introduced the PRF algorithm and compared their implementation with RF using synthetic data. As for this paper, we carried out the same experiments but using real data sets and adding a few more methods such as cross validation to get an accurate performance estimation [12] and applied different imputation methods. The results of the experiments are gathered in terms of comparing the prediction performance, the influence of different threshold values, being a new hyperparameter introduced by PRF, on the different imputation methods, and lastly the computational time for different number of trees.

This paper is structured as follows: in section 2, we will explain the RF algorithm, its probabilistic version and some performance measures in greater detail. Then, in section 3, we will describe how we carried out the experiments with the standard RF and PRF and present the results in the following section. After that, we will evaluate the results in the discussion section and finally provide a conclusion and recommendation for future work.

2 Methodology

An ensemble is, arguably, an effective method to improve accuracy of a classifier. Random Forest (RF) is one example of such a method, where it forms an ensemble of decision trees to yield a prediction outcome by means of a majority vote. However, as mentioned previously, RF treats the features and labels as deterministic values. Thus, it might not necessarily have a good performance when applied to a noisy data set. To account for this noise, or uncertainties of the data, we can treat the features and labels as probability distribution function, which is done by the Probabilistic Random Forest (PRF).

- *Zahra Putri Fitrianti is with University of Groningen, E-mail: z.fitrianti@student.rug.nl.*
- *Codrut-Andrei Diaconu is with University of Groningen, E-mail: c.a.diaconu@student.rug.nl.*

2.1 Random Forest

Random Forest (RF) is a combination of trees (predictors), such that each tree uses a random subset of features and a random subset of the training data points [5]. Then, these trained trees are formed into an ensemble with a large number of trees. They will then yield a single prediction output which is based on combining the outputs of each tree by means of majority vote.

When applied to a binary classification task, i.e. there are only two classes $C \in \{0,1\}$, the algorithm searches for the "best split" by taking into account both the features and a constant threshold that will result in the best separation between the two classes [14]. Hence, to assign a class to a new example, it needs to be propagated through the tree and in each node we need to determine the split. To do so, we can consider the Gini impurity which is measured as the probability of misclassifying a new example, if it is randomly classified based on the distribution of the class labels in the data. Subsequently, this is a good measure to use because it minimizes misclassification error. In mathematical notation, this can be expressed as:

$$Gini(t) = 1 - \sum_{i=0}^{c-1} p(i|t)^2 \tag{1}$$

where $p(i|t)$ represents the probability of class i in node t and c is the number of class.

The Gini impurity is used as a condition for the new example, to either propagate to the right node or the left node recursively until it reaches the leaf node. When it does, the algorithm will stop and the new example will be assigned to one of the two classes. The prediction of the label is done by taking a majority vote among the trees, or, take the class which has the highest probability in the leaf. By doing so, it can be ensured that the accuracy will be improved because typically, a single tree tends to overfit during training and hence not have a good generalization performance. This problem is overcome by combining multiple trees in Random Forest as it generalizes well to novel examples and has a higher accuracy compared to a single tree [5].

2.2 Probabilistic Random Forest

Probabilistic Random Forest (PRF) is a modified version of the RF method but it treats the features and labels as probability density functions thereby taking into account the uncertainties [14]. In a classification task, usually there are two kinds of uncertainties, i.e. feature uncertainty and label uncertainty, and PRF is able to handle both.

Since the PRF is based on the standard RF method, it follows a similar idea. Given a new unlabelled example, we can try to classify this in the PRF by propagating through the probabilistic tree. Thus, the algorithm also searches for the "best split" for which the new example should propagate to. In order to determine the best split, the PRF takes into account the combined probability for an example to take all the turns that led to a certain node, perhaps deep in the tree, from the root. The example then keeps being propagated to until it reaches a certain threshold, which is a new hyperparameter that is introduced by the PRF algorithm which controls when to stop propagating in the probabilistic tree.

As done in the standard RF, the best split is determined by calculating the Gini impurity. However, since we have class probability, a fraction of examples in the data in some given class is now a random variable [14], which consequently affects the calculation of the Gini impurity. In this case, Eq.(1) is rewritten to

$$\bar{Gini}(t) = 1 - \sum_{i=0}^{c-1} \hat{p}(i|t)^2 \tag{2}$$

where $\hat{p}(i|t)$ is the expectancy value of class i in node t and c is the number of class. $\hat{p}(i|t)$ is given by the following equation

$$\hat{p}(i|t) = \frac{\sum_{i \in t} \pi_i(t) \cdot p(i|t)}{\sum_{i \in t} \pi_i(t)} \tag{3}$$

In Equation (3), $\pi_i(t)$ refers to the probability of propagating through the tree at node t. Let $n \in R$ and $n \in L$ denote all the nodes that belongs to the right and left branch, respectively, the formula to calculate $\pi_i(t)$ for any arbitrary node t is given by

$$\pi_i(t) = \prod_{n \in \mathrm{R}} F_{i,k_n}(\chi_n) \times \prod_{m \in \mathrm{L}} (1 - F_{i,k_m}(\chi_m)) \tag{4}$$

where $F_{i,k}(\mathrm{X})$ denotes the cumulative distribution function given by

$$F_{i,k}(\chi) \equiv P_r(X_{i,k} \leq \chi) = 1 - P_r(X_{i,k} > \chi) \tag{5}$$

for $X_{i,k} \sim \mathrm{N}(x_{i,k}, \Delta x_{i,k}^2)$, provided that $x_{i,k}$ is the measured feature value and $\Delta x_{i,k}^2$ is its corresponding uncertainty. Referring back to Equation (3), this basically means that the probability for any object i to reach node n in the tree, is calculated by multiplying the combined probability of propagating to both left and right nodes until it reaches node n. To put it simply, it can be said that PRF does take into account the probabilities of going to the right and left nodes while propagating the tree from a certain node. If it turns out that the data contains missing values, then PRF assigns equal probability for propagating to both directions and use this value for the next node until it reaches a certain threshold.

Since the algorithm searches for the split in which the two resulting nodes are more homogeneous than its parent's, we need to define a cost function which should be minimized during the training process. This is defined as the weighted average of the modified Gini impurities of the right and left nodes, i.e.

$$e = Gini(r) \times \frac{\sum_{i \in (t,r)} \pi_i(t,r)}{\sum_{i \in t}} + Gini(l) \times \frac{\sum_{i \in (t,l)} \pi_i(t,l)}{\sum_{i \in t}} \tag{6}$$

where $\pi_i(t,r)$ and $\pi_i(t,l)$ are the probabilities of propagating to the right or left node respectively.

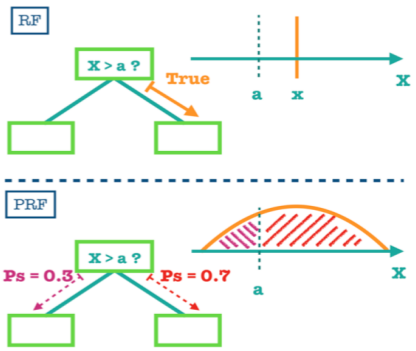


Fig. 1. taken from [14], it shows the difference between RF and PRF in terms of finding the "best" split. It also depicts the fact that RF treates features and labels as deterministic values whereas PRF estimates their probability density function (pdf).

To sum up, the main differences between PRF and the standard RF are: PRF treats features and labels as probability density functions, it uses combined probability to determine to which node it should propagate to, and the cost function that is minimized by the algorithm. As for determining the best split, the procedure remains the same as in the standard RF, but the calculation for Gini impurity is slightly changed since it now uses expectation value [14]. As an illustration, this is shown by Figure 1.

2.3 Performance Measures

It is a common practice in machine learning, more so in classification tasks, that the quality of the prediction outcome is evaluated. In a broader context, this is done because us, as users, would like to know how reliable the resulting prediction output is. This can be done by

means of calculating some performance measures or evaluation metrics which are simply a set of statistical measures that can be used to quantitatively describe the predictive performances in different aspects under different conditions [8].

Generally, the evaluation metrics are applied during both training, to optimize the classifier, and testing, to measure how reliable the prediction output is when tested with novel examples. Pertaining to binary classification problems, the evaluation metrics are defined based on a confusion matrix [7]. In a confusion matrix, the quantities TP and TN refer to the number of examples that are correctly classified as positive and negative respectively. By contrast, FP and FN denote the number of examples that are incorrectly classified as positive and negative respectively. These four metrics serve as the basis to compute the following performance measures.

Accuracy is formally defined as the ratio of correctly classifying all examples. Its formula is given by:

$$Accuracy = \frac{(TP+TN)}{(TP+TN+FP+FN)} \tag{7}$$

F1-score is another performance measure which is typically used for unbalanced data sets. It represents the harmonic mean between precision and recall [7]. This is calculated as:

$$F1-score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{8}$$

where

$$Precision = \frac{TP}{(TP+FP)} \tag{9}$$

$$Recall = \frac{TP}{(TP+FN)} \tag{10}$$

Receiver Operating Characteristic (ROC) describes the relation between sensitivity and specificity. Sensitivity is simply the true positive rate (TPR) which is also defined by Eq.(10) and it is represented by the y-axis in the ROC curve. In simple terms, TPR = sensitivity = recall. Specificity is the true negative rate (TNR) which is mathematically defined as:

$$Specificity = \frac{TN}{TN+FP} \tag{11}$$

In the ROC curve, the x-axis represents the false positive rate (FPR) which is simply

$$FPR = 1 - Specificity \tag{12}$$

The ROC curve represents a combination of pairs of (FPR, Sensitivity) which are calculated by using different values for decision threshold. Generally, we expect the curve to be close to the top-left corner as it indicates that the classifier or predictor has a good classification performance. If the curve is close to the diagonal, this implies that the predictions are based on or close to random guesses [8].

Area Under ROC Curve (AUC) measures the performance of the predictor by one value which is computed based on the area under the ROC curve previously discussed. The AUC of an ROC curve indicates the deviation from random guessing. In other words, it computes the probability that a randomly selected positive example gets higher scores than a randomly selected negative example [8].

3 Experiments

We carried out several experiments to compare the standard RF and PRF, more specifically in terms of its performance and computational time. We used the RF implementation from python's `scikit-learn` [11] library and the implementation of PRF which can be found in [13]. The experiment was designed as follows:

1. Apply 10-fold cross validation to get different training and testing subsets.
2. Apply three imputation methods (mean imputation, median imputation, KNN imputation) to both training and testing subsets.
3. Train both RF and PRF with the imputed training sets and test using testing sets. Even though PRF deals with missing values by design, we applied imputation to both RF and PRF such that they are tested or compared under the same condition.
4. Calculate the performance using the several metrics (accuracy, F1-score, ROC AUC) and calculate the computation time.

Each step of this experiment is described in detail in the following subsections.

3.1 Hyperparameter settings

In our experiments we considered combinations of the following hyperparameters:

- Number of trees = 10, 25, 100
- Threshold for PRF = [0.01, 0.05, 0.10, 0.25, 0.50, 0.75, 1.0]
- The value of k in k-nearest neighbour imputation = 7, using Euclidean distance.
- The value of n in n-fold cross validation = 10

3.2 Data sets

We deliberately chose data sets which have missing values since it can be considered as a type of noise. For the experiment, we used the following data sets:

Pima Indians Diabetes Database [9] This data set is used to predict whether a person has diabetes or not by taking into account several features such as Glucose, Blood pressure, Skin thickness, Insulin, BMI and Age. From Figure 2 below, we can see that Insulin level has the highest missing value fraction, being 48.7%. On average, this dataset has 14.4% of missing values. Furthermore, referring to the class label distribution, it can be said that the data set is a bit imbalanced.

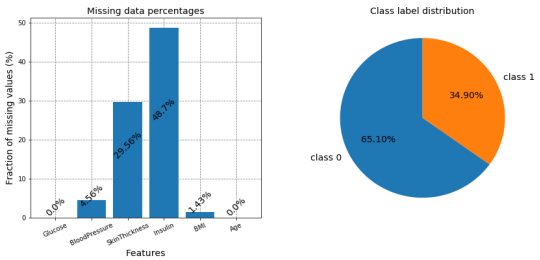


Fig. 2. Plots of missing data fractions (*left*) and class label distribution (*right*) of the Pima Indians Diabetes Data Set. In the class label distribution plot, class 0 refers to not having diabetes whereas class 1 refers to having diabetes.

Mammographic Mass Data Set [15] This data set is used to predict whether a mammographic mass lesion is benign or malignant based on six features: BI-RADS, patient's age, shape, margin and density. Figure 3 shows the missing data fractions and the class label distribution, respectively. In the left plot, it can be seen that the feature Density has the highest missing value percentage, being 7.91%, and the data has on average 3.372% of missing values. Moreover, in the class label distribution, we can see that each class has roughly the same number of examples in the data set.

Thyroid Disease Data Set [16] Looking closely at Figure 4, it can be seen that this data set has on average 11.75 % of missing values. Furthermore, referring to the class label distribution plot, it can be deduced that the data is very imbalanced where class 0 fills the majority of the data set while class 1 only accounts for 4.75% of the data. The original dataset is composed of 21 binary attributes, and 7 continuous features (patient age and six laboratory test results) but during our experiments we used only a subset of five laboratory test results as described in Section 9.3 of [6].

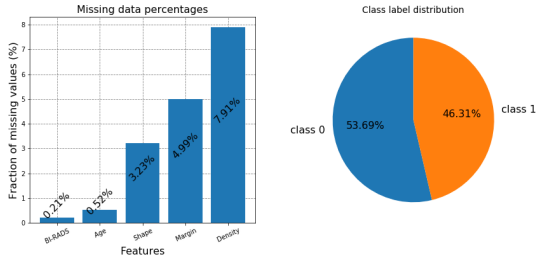


Fig. 3. Plots of missing data fractions (*left*) and class label distribution (*right*) of the Mammographic Mass Data Set. In the class label distribution plot, class 0 refers to benign class whereas class 1 refers to malignant class.

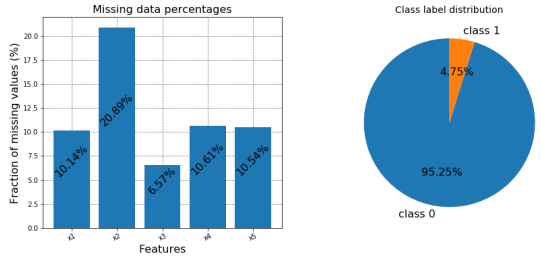


Fig. 4. Plots of missing data fractions (*left*) and class label distribution (*right*) of the Thyroid Disease Data Set. In this data set, class 0 refers to the healthy patients whereas class 1 refers to the patients who are diagnosed as sick.

3.3 N-fold Cross Validation

Cross Validation is a technique where we randomly split the data into disjoint subsets, i.e. training set and testing set. Typically, cross validation is used for model selection and it aims to avoid overfitting where the data is too well-trained that it has a poor generalization ability when tested with unseen data. Furthermore, to make sure that the model is trained with different subsets, we can apply one of its variant which is called N-fold cross validation. Here, the data is split into N disjoint subsets of the same size. We perform this for N number of iterations such that in each iteration, we obtain different training and testing sets, and eventually we use $N-1$ subsets for training and one subset for testing or validation. We used the `StratifiedKFold` cross validation from the `scikit-learn`'s library. To add a little bit more context, stratification ensures that each fold is a good representative of the entire data set [12].

3.4 Imputation

Missing values is a very common situation that occurs in most scientific research fields such as Medicine, Astronomy, and Computer Science. There are a number of reasons which caused missingness, some of which are incomplete surveys, wrong information, incorrect measurements, and many more [10]. In classification task, it is important to have a complete data set because the algorithms usually assumes that the training data is complete, yet in reality, it could consist of missing values.

For this experiment, as described in subsection 3.2, we chose data sets which have missing values. However, since the RF implementation from python's `scikit-learn` library does not deal with missing values by design, therefore, we have to first apply imputation method. As stated previously, we used three imputation methods namely Mean Imputation, Median Imputation, and KNN Imputation.

3.4.1 Mean Imputation

This imputation method is based on a statistical approach, where each missing value of a feature is imputed by the mean (or average) of the observed values for that feature [4]. Even though this method is very commonly used and, surprisingly enough, has given good results for classification problem, unfortunately, it could potentially lead to changing the distribution of the data entirely since it underestimates the variance and can decrease a correlation coefficient due to disregarding interaction effects of the variables [1].

3.4.2 Median Imputation

This method is also based on a statistical approach, except that instead of substituting the missing values with the mean, they are replaced by the median (the middle point in the data). It was originally introduced to assure robustness since the mean is usually affected by outliers in the data [1]. Formally speaking, this method has a similar impact as the mean imputation method, in the sense that it could distort the distribution of the data. If the data has a skewed distribution, then this method is good to use. Similar to mean imputation, this method could potentially lead to wrong results as it completely disregards the correlation between the variables in the data.

3.4.3 KNN Imputation

The idea behind this method is to find the k-nearest neighbours and replace the missing values by its mode or average for qualitative features or quantitative features, respectively [1]. This method is different than the previous two because it is based on a non-parametric classifier which takes into account the explicit model of the data. The advantages of using this method are: it is able to predict both qualitative and quantitative features. It can also easily treat instances with multiple missing values. More importantly, it takes into account the correlation structure of the data, which subsequently overcomes the problem that mean imputation method has. Although it seems like KNN imputation method is the better option among the three aforementioned methods, much like the nature of the algorithm, this method is highly dependent on the distance metric that is used to determine the nearest neighbour. There are several distance measures that can be used such as Euclidean, Mahalanobis, Manhattan, and so on. Each of this metric has a different effect to the algorithm. In addition, it is also computationally expensive as it searches through the entire dataset.

4 Results

The following results are obtained from conducting the experiment and are used to compare the standard RF and PRF in terms of performance and computation time. Note that in the following two subsections, we only present the results obtained from running the experiment with 25 trees (giving the space limits) whereas for the computation time comparison we also included 10 and 100 because we could combine them in one figure. It is important to notice that the results based on 25 trees are similar to those using 100 in all our experiments.

4.1 Performance Comparison

Figure 5 shows three plots for the three data sets during training (top plot) and testing (bottom plot). Each plot shows the average accuracy, F1-score, and ROC AUC over the 10-fold cross validation, obtained during training and testing for both RF and PRF, using the KNN imputation method, 25 trees and PRF threshold = 0.05.

Looking closely at the top plots in Figure 5, it can be seen that during training the standard RF generally has a high accuracy and better F1 scores with small standard deviations. On the other hand, PRF also has high accuracy, but lower than RF on average over the three data sets, and slightly lower F1 scores. In addition, the average ROC AUC are also quite high which implies that RF and PRF both have a good classification performance, but in some cases RF is better, e.g. in the mammographic data set.

Conversely, we can see in the bottom plot in Figure 5 that during the testing phase which is more relevant w.r.t. generalization, the standard RF has lower values in all performance metrics, i.e. accuracy, F1 scores, ROC AUC, than PRF. By contrast, PRF still has high values during testing, despite of them being lower than what was previously obtained during training. On the same note, the average standard deviation also increases for all metrics and for both RF and PRF. As a general intake, these results show that PRF does perform better than

the standard RF and it does improve the accuracy and classification performance during testing.

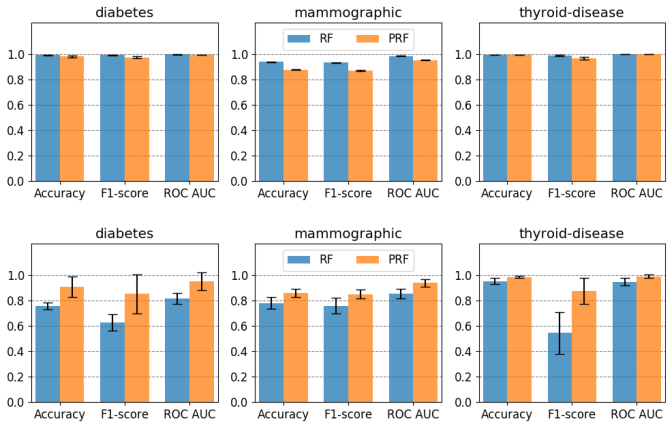


Fig. 5. The three plots above show the average accuracy scores, average F1-score, as well as average of ROC AUC of training (*top*) and testing (*bottom*) RF and PRF using KNN imputation method. The error bars represent the standard deviation over the 10 folds.

4.2 The influence of PRF Threshold and RF Imputation Method

Atop of comparing the classification performance, we also compared the results for different imputation methods with respect to multiple threshold values. In this experiment, we compared all three imputation methods that are applied to RF with the KNN imputation method applied to PRF. We only present the result for KNN imputation on PRF since the other two methods give similar results. More than that, PRF yields similar performance even without imputation. It is important to note that we included only the F1 scores since it is the most appropriate measure when we have imbalanced data sets. In addition, the results are displayed as lines for helping the comparison.

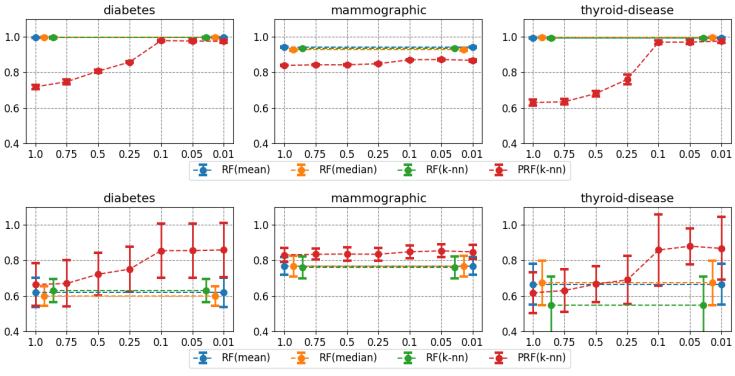


Fig. 6. The three plots above show the average accuracy scores, average F1-score, as well as average of ROC AUC of training (*top*) and testing (*bottom*) RF and PRF using KNN imputation method. Note that the error bars correspond to the standard deviation over the 10 folds.

From the top plot in Figure 6, we can see that during training, generally RF has stable and high F1-scores for the different imputation methods. By contrast, PRF has more of an increasing trend of the F1-score for different threshold values. Except for the Mammographic data set where it is more or less stable with minor fluctuations for small threshold values.

On the other hand, by referring to the bottom plot in Figure 6, we can see that RF, in turn, has constant and lower F1 scores but higher standard deviations, compared to the training plots. Looking closely at the plot for thyroid-disease data set, it can be seen that the standard deviation becomes very large during testing, and increases in general. Although it is true that the standard deviation is still quite large, especially for diabetes and thyroid disease data sets, it still yields a better performance than RF during testing.

4.3 Time Complexity

Figure 7 shows the plots of the execution time of RF and PRF, with respect to different number of trees (10, 25, 100) and different threshold values. In this part of the experiment, we computed the average time during training over the 10 folds. As we can see in the figure below, the PRF algorithm always takes more time to run. Moreover, as the threshold values decrease, the average computation time increases. In other words, the threshold is inversely proportional to the average computation time. Furthermore, it is also important to note that in the case of RF, the ratio is almost constant for different values of threshold, simply because RF is independent of this.

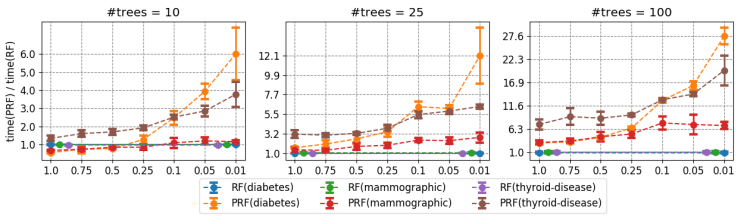


Fig. 7. The three plots above show the computation time ratio with respect to different threshold values for different number of trees. In these plots, the error bars are the standard deviation over the 10 folds.

5 Discussion

We compare the standard RF and PRF in terms of its performance and computation time based on the results presented in the previous section. Additionally, we compare the influence of the different imputation methods on RF with one imputation method (KNN) on PRF with respect to different threshold values.

5.1 Classification Performance

Analyzing the testing results from Figure 5, it can be seen that for all three data sets, PRF outperforms RF especially in terms of F1 scores. It is also important to note that the smallest difference in the performance of the two algorithms occurs in the case of the Mammographic Mass data set which has the smallest fraction of missing data whereas, for the other two data data sets, PRF outperforms RF by a large margin. Furthermore, given that PRF yields better F1 score, this implies that it deals with imbalanced data sets better than the standard RF, specifically when considering the class distribution of thyroid-disease dataset which can be seen from Figure 4.

Moreover, as was previously mentioned, the same imputation method was used for both algorithms, in spite of the fact that PRF already has the capability to deal with missing values. In our experiments, PRF gives similar performances even if we do not impute the data beforehand. This gives another advantage which is avoiding the problem of dealing with missing values.

Comparing the testing results with the training ones, we can observe that RF usually gives better results compared to PRF, which implies that there is an indication of overfitting. This observation is based on the fact that PRF estimates the probability density function (pdf) of the features which is usually a more robust way to model the data than by just using the deterministic values.

5.2 The Influence of PRF Threshold

As explained in [14], a PRF threshold close to one means that we stop the propagation through inferior nodes. This means that PRF should have similar performance to the standard RF algorithm. To evaluate

the influence of this new hyperparameter, it can be clearly seen in Figure 6 that in most of the cases, RF gives better F1 scores on training but worse for testing, which also confirms the observations made previously about overfitting. One interesting remark is that the average F1 scores of PRF are quite similar during training and testing and they roughly follow the same trend. Regarding the influence of the threshold itself, the results confirm our expectation that the lower threshold increases the odds of a good prediction. Finally, the two plots also show that for a threshold below 0.1, the results are approximately constant, thus a value of 0.05 seems to be a good choice.

On the same note, these two figures also include the results of RF for all three imputation methods. Except for the thyroid dataset, the results are quite similar so RF is not too sensitive to how the data was imputed, yet we can not generalize that this will happen for other (more complex) imputation methods. Regarding PRF, it gave almost the same performance for all the imputation methods and also on the non-imputed data, which is why we only included one case (i.e. k-nn) in the results.

5.3 Time Complexity

Propagating a sample further on both branches also needs more computation time. The time complexity for traversing a balanced tree is $O(log(n))$, where n is the number of nodes. If we use a very low threshold in PRF, this implies that we visit all the nodes, in which the time complexity becomes $O(n)$. Of course, these measures are purely theoretical and we ignore the fact that PRF also needs extra time for estimating the pdf and computing the probabilities for going left or right. More importantly, the number of splits, which gives the number of nodes in the final tree, is always unknown before training and dependent on the data.

Giving these difficulties, we decided to analyze the computation time averaged over many runs, as shown by Figure 7. Since we are interested only in comparing the two algorithms, we considered the RF execution time as a reference, therefore it has the value one on the y-axis, and for PRF we compute the ratio between its average execution time and the RF's. Notably, it is also impractical to use the actual execution time since this depends on the machine but the ration obtained by averaging over the 10 folds should at least give an estimation of actual complexity of PRF compared to RF. First, as we expected, a low threshold brings a high computation time. Furthermore, for a large number of trees, the complexity of PRF becomes an impediment, for example when using 100 trees, PRF is almost 30 times slower in case of the diabetes dataset and this amount of trees is frequently used in practice (for instance, this number will become the default in the following versions of `scikit-learn` RF implementations). Of course, as was previously mentioned, this depends on how predictable the data is, for instance, in the case of the Mammographic Mass data set, PRF is only 6 times slower for 100 trees.

6 Conclusion

Random Forest (RF) is an ensemble method which consists of multiple decision trees and performs classification by means of majority voting. Probabilistic Random Forest (PRF) is its variant, which estimates the probability density function (pdf) of the features and labels and use these values instead of the deterministic values as done in the standard RF. The main difference between RF and PRF is, since PRF considers the pdf of the features, therefore, during the split, it takes into account the probabilities of propagating to both left and right branches, and it keeps going until it reaches a certain threshold. This way of splitting a sample brings a new hyperparameter – a threshold which controls when to stop the propagation through the tree. It was shown that this threshold plays a very important role and it is strongly correlated to the final prediction performance.

A few experiments were carried out to compare RF and PRF in terms of its performance, influence of imputation method with respect to different threshold values and the computation time. The obtained results indicate that PRF generally has a better performance than RF. This was shown by calculating several performance metrics namely accuracy, F1-score, and ROC AUC where PRF usually has higher scores during testing. Furthermore, we observed that during training, RF has very high values of the aforementioned metrics, whereas during testing it gives lower values. From this observation, it can be concluded that there is an indication of overfitting which is done by RF during training. On the other hand, PRF has high values for all performance metrics during both training and testing, with an exception of the results obtained from the mammographic data set. It can be deduced from here that PRF is a better classifier which has a good classification performance and high accuracy when the data contains a significant amount of missing values. However, one major disadvantage of PRF is in terms of computation time which is correlated to the propagation threshold; a very low value makes PRF too slow in some cases.

For further work, we will also consider a complete dataset with a more complex structure and evaluate the algorithms on it. Following the example from [14], we can add missingness in a structured way and then evaluate the impact of the quantity of missing values in terms of both performance and execution time. Another more complex imputation mechanisms can also be considered for RF. Moreover, we can consider other types of noise although this forces us to only inject it since usually in practice it is difficult to estimate the noise from the existing data without external information about it.

Acknowledgements

The authors wish to thank Michael Biehl for his reviews and inputs on the experiments that we carried out, and also fellow colleagues for their reviews.

References

[1] E. Acuna and C. Rodriguez. The treatment of missing values and its effect on classifier accuracy. *Journal of Classification*, pages 639–647, 01 2004.

[2] J. E. T. Akinsola. Supervised machine learning algorithms: Classification and comparison. *International Journal of Computer Trends and Technology (IJCTT)*, 48:128 – 138, 06 2017.

[3] E. Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2014.

[4] C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[5] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct 2001.

[6] P. J. García-Laencina, J.-L. Sancho-Gómez, and A. R. Figueiras-Vidal. Pattern classification with missing data: a review. *Neural Computing and Applications*, 19(2):263–282, 2010.

[7] M. Hossin and S. M.N. A review on evaluation metrics for data classification evaluations. *International Journal of Data Mining & Knowledge Management Process*, 5:01–11, 03 2015.

[8] Y. Jiao and P. Du. Performance measures in evaluating machine learning based bioinformatics predictors for classifications. *Quantitative Biology*, 4(4):320–330, Dec 2016.

[9] Kaggle. Pima Indians Diabetes Database. `https://www.kaggle.com/uciml/pima-indians-diabetes-database`. [Online; accessed 08-March-2019].

[10] R. Pan, T. Yang, J. Cao, K. Lu, and Z. Zhang. Missing data imputation by k nearest neighbours based on grey relational structure and mutual information. *Applied Intelligence*, 43, 05 2015.

[11] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[12] P. Refaeilzadeh, L. Tang, and H. Liu. Cross-validation. *Encyclopedia of Database Systems*, 532538:532–538, 01 2009.

[13] I. Reis and D. Baron. Probabilistic Random Forest (PRF). `https://github.com/ireis/PRF`. [Online; accessed 12-April-2019].

[14] I. Reis, D. Baron, and S. Shahaf. Probabilistic Random Forest: A Machine Learning Algorithm for Noisy Data Sets. *AJ*, 157:16, Jan 2019.

[15] U. M. L. Repository. Mammographic Mass Data Set. `http://archive.ics.uci.edu/ml/datasets/mammographic+mass`. [Online; accessed 08-March-2019].

[16] U. M. L. Repository. Thyroid Disease Data Set. `http://archive.ics.uci.edu/ml/datasets/thyroid+disease`. [Online; accessed 08-March-2019].

Comparison of data-independent Locality-Sensitive Hashing (LSH) vs. data-dependent Locality-Preserving hashing (LPH) for hashing-based approximate nearest neighbor search

Jarvin Mutatiina, and Chriss Santi

Abstract— Optimizing nearest neighbor search for high dimensional data is a popular research area with many studies centered around how approximation with hashing can be alternative in accelerating the search. The interest stems from the fact that nearest neighbor search can be complex for such data requiring traversal of all data point to find similar points ; which requires intensive computational resources and is generally slow. This is especially important for applications like document search and image retrieval from large repositories. Hashing techniques accelerate search by limiting the search space to a definite bound based on certain criteria. In this paper, we discuss how two hashing techniques - Locality Sensitive Hashing(LSH) and Locality Preserving Hashing(LPH) as approximates for nearest neighbors search. We compare the use of LSH and LPH and discuss usage scenarios for when the respective techniques are relevant. The comparison is based on the accuracy of the search and query time. This will further explore the trade off between the two aspects- accuracy and query time. We also make suggestions for future work.

Index Terms—Nearest Neighbor search, Locality Sensitive Hashing, Locality Preserving Hashing, Hashing -based approximation for nearest neighbor search.

1 Introduction

Determining the most similar data in relation to a query is an important aspect for many applications like image retrieval, document search, pattern recognition, data compression etc. With high dimensional histogram data like text and images, performing a nearest neighbor search using traditional methods like *k-d* trees becomes complex[3] and is likened to a brute-force linear search; as each data element might have to be traversed. This can be computationally intensive and time consuming e.g. for queries over the internet and this brings a need for novel techniques to mitigate this. Approximation of neighbors with hashing functions is identified as a viable alternative to determining the most similar neighbor while eliminating the poor performance demonstrated by a linear search on high dimensional data[3]. The hashing approximations limit the search space by reducing the dimensionality of the data into a finite length hash value and then grouping the similar values together. This drastically accelerates search especially on histogram data(e.g images and text) that has semantic similarity. This makes the techniques much suited for high dimensional data as comparison is not across the dimensions of each data sample. The result to the query may not be accurate or exact but it is intuitively guaranteed that the approximate result is within a reasonable error bound. In addition, approximates greatly contribute to improved computational speed and storage reduction for nearest neighbor search[3].

Several hashing algorithms have been proposed to aid the approximation for nearest neighbor search in high dimensional data and in this research we focus on comparing two top contenders - Locality Sensitive Hashing(LSH) and Locality Preserving Hashing(LPH). Hashing translates each data point into a compact, finite length hash key that points to the data values. This compactness is low dimensional and enforces the similarity search with a linear time complexity[13]. With LSH, similar hash keys are stored in the same bucket and approximation with similarity search happens by hashing the query point and retrieving the other points stored in the same bucket. Also, the probability of collision is higher for similar points than for points that are far apart[3]. Collision might be avoided in generic hashing but LSH relies on this phenomenon to group similar data points. The downside to LSH is that the hashing is random and independent of the data hence similar data might end up in different buckets[1]. This affects the results of the similarity search. On the other hand, LPH is a data dependent hashing function that maintains the neighborhood structure of the input data during hashing i.e. hash keys are generated based on the structure of the data. The collection of hash keys will have a representative similarity structure as the original data. This is an improvement towards similarity search with higher accuracy as the probability of grouping similar data points is even higher than LSH. Despite, the improvement, LPH is significantly slower in terms of query time and disk access[3]. This leaves a trade-off between query time and accuracy of results of the hashing-based approximation, which also explains the usage scenarios for which the two functions are most relevant.

Besides the hashing techniques of interest, there exists numerous algorithms that work to optimize the nearest neighbor search. Other data dependent techniques include; Semantic hashing[12] which works well for document retrieval with a higher precision and recall than LSH. Isotropic hashing[7] that learns different projection dimensions with equal variance to cater for larger variance dimensions that carry more information, and Spectral Hashing that relies on principal component analysis and constructs the hash codes with a subset of eigenvectors of the Laplacian of the similarity graph [15]. LPH however, still performs considerably better than these techniques. On the other hand, alternative data independent techniques include; Kernelised Locality-sensitive Hashing(KLSH)[9] which is widens the LSH for accessibility to arbitrary kernel functions without knowledge of the underlying feature space and Binary Reconstructive Embeddings(BRE)[8] which also improves on the random projections used in LSH.

Our investigation focuses on how and when to use Locality Sensitive Hashing and Locality Preserving Hashing as approximations for hashing-based nearest neighbor search plus the influence that data (in)dependence has on the approximations. In this paper, we explore both hashing functions and examine their relevance, usage scenarios and assumptions related to the mentioned functions. We analyze results of experiments from published research by Yi-Hsuan Tsai and Ming-Hsuan Yang[13] on image datasets - CIFAR-10 and CIFAR-100. The results are evaluated based on accuracy metrics of precision and recall. The paper also discusses the relevance of the hashing techniques in relation to high performance approximate nearest neighbor

- *Jarvin Mutatiina, E-mail: j.mutatiina@student.rug.nl*
- *Chriss Santi, E-mail: c.o.santi@student.rug.nl*

search. Finally, we present a conclusion and suggestions for future work.

2 METHODOLOGY

In this section, we discuss some recurring concepts that will be mentioned in this paper.

2.1 Nearest Neighbor Search

Nearest neighbor search is the concept of finding the closest/most similar point to a query in a given dataset. It can also be looked at in terms of the K nearest neighbors, which returns the K nearest elements in regards to a given query. This similarity can be measured with several distance metrics like Jaccard similarity, hamming distance, euclidean distance etc.

Hamming distance is a popular metric used that relies on the concept of; a d-dimensional hamming space with strings of length d with each point $x \in H^d$. The hamming distance between two points is the number of positions at which the two points' corresponding strings differ in the hamming space. This distance metric also has a corresponding time complexity of $O(d)$.

Jaccard similarity is the ratio of the size of intersection of two given sets of data(can be the query Q and data points X) to the size of their union i.e $|Q \cup X| / |Q \cap X|$. Figure 1 illustrates this concept [11].

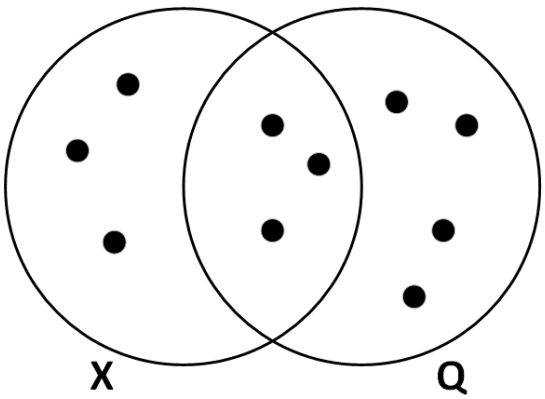


Fig. 1. Jaccard similarity of 3/10

For example in document search over the internet, Jaccard similarity drives the similarity search by facilitating discovery of textually similar documents based on character similarity not similar meaning. In events of duplicates or plagiarism, the similarity search performs exceptionally well. And the same concept is employed in nearest neighbor search.

Nearest neighbor search is useful in many applications like image retrieval, document search. In this case, we are searching not necessarily for a duplicate but also the most similar in terms of content and context.

However, with high dimensional data, a nearest neighbor search is likened to a linear search. This is because for sufficient search to happen, all data examples have to be traversed and compared individually with the query. This makes the time complexity equivalent to O(N) with N being the size of the dataset. This rises a need for higher performing searching criterion with better query time and uses less computational resources. Approximation with hashing functions reduces the complexity to sub-linear by reducing the size of the comparison space. An approximate can be defined in euclidean and or hamming space [14] as; $(1+\in)$ where for a query q, we find a point x with dist$(q,x) \leq (1+\in)$ dist(q,x^*).x^* is the actual nearest neighbor. Approximation may not guarantee a completely accurate result but is a much faster alternative as it returns the most relevant points in relation to a query.

2.1.1 Hashing based Approximate search

The main idea in hashing based approximate search is to eliminate the need to compare with every data sample in a high dimensional dataset in order to find the nearest similar neighbor for a given query. Hashing can be precisely defined as a mapping of data points to a compact definite code - hash keys/codes/values. These hash codes take an arbitrary long input and translate into a code of a definite length that exist in a low dimensional definite space.

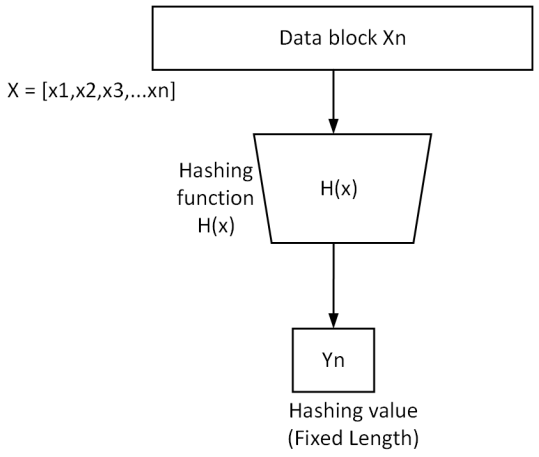


Fig. 2. How generic hashing works

Given a dataset X = $x_1, x_2, ..., x_n$[symbol] $R^{(}nxd)$, the basic idea of hashing is to map each point x_i to a suitable K- dimensional binary code y_i -1,+1 with K denoting the code size[16].

This can be also accurately represented as $y = H(x)$ with H as the hash function.
Hashing occurs in two stages where data points are first projected in a smaller dimension - *projection stage* and then the projected values are quantized into hash codes - *quantization stage*. Hashing aids approximation by limiting the search space and hence accelerating the search. This introduces the concept of hashing based approximate search.

2.2 Locality Sensitive Hashing

Locality sensitive hashing is a hashing technique that relies on the principle of maximizing of probability of collision for objects that are similar as compared to those that are far apart. This can be referred to as "sensitivity". In this context, collision refers to mapping multiple hash codes to the same bucket. As opposed to generic hashing that avoids collision, LSH relies on this phenomenon to aid the nearest neighbor search. As elaborated by Andoni et al[1],sensitivity can be formally described as;

$$\rho = \frac{log(1/p1)}{log(1/p2)} \tag{1}$$

where;
p1 is the collision probability for nearby points
p2 is the collision probability for points that are far apart and
ρ is the measure of how sensitive the hash function is to distance.

The space complexity for the above definition 1is $O(dn + n^{1+\rho})$, query time complexity- $O(n^{\rho})$ [2].

With the ability to quantify sensitivity, this technique can imposed over several hashing functions which store hash codes in buckets/

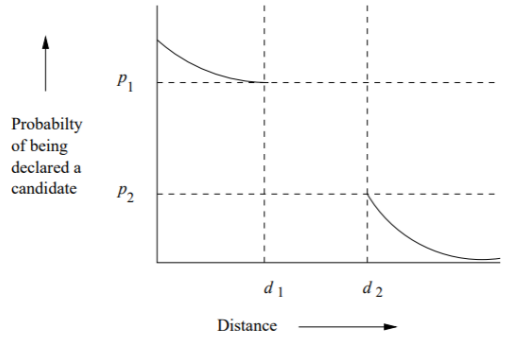


Fig. 3. Shows illustration of sensitivity ρ [11]

repositories and in essence on query, similar objects will be placed to the same bucket. This is illustrated in figure 4. Therefore, given a query, LSH has the ability to approximate similar data points based on the bucket of the query.There can be constraints to this technique as the hashing is conducted randomly. This also explains the phenomenon of data independence and how the hashing function does not maintain the similarity structure of the input data based on any distance metric. This structure is not represented in the output hash code.

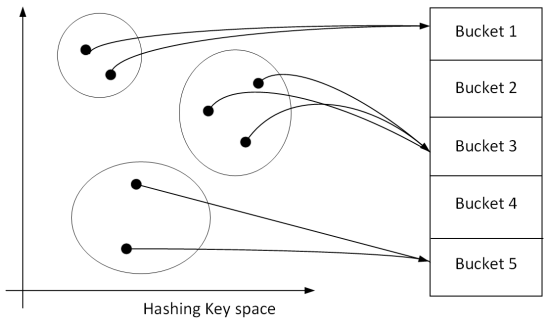


Fig. 4. Hashing of similar data samples into the same buckets

Random hashing or projections of data leaves the possibility that similar objects might end up in different buckets and this adversely affects the nearest neighbor search. Despite this being a shortfall, the randomness of LSH gives it a fast query time, a small memory footprint[1] and a fair approximate in relation to the traditional nearest neighbor search. This further introduces the query time and accuracy trade-off which is highly dependant on the scenario or application. Would a user rather wait for an accurate results or get an approximate in timely manner?

2.2.1 Implementing LSH

To elaborate LSH further especially for document/text search, we can look at it in 3 broad steps;

- **Shingling** This involves splitting a given document into a set of phrases or words. With the fragments that exist in these sets, we are able to lexicographically compare with other sets.
  The shingle length k should be large enough so that the probability of any given k-shingle appearing in any given document is relatively low. For a given document D ={dgfhskrbf} and K =3, the shingles for D are dgf,hsk,rbf.

- **Minhashing** Shingling generates large sets of groupings of the data and in this step, we abstract these large sets with compact representation. To do this, we first construct a *characteristic matrix* that corresponds to the occurrence of certain elements in all the sets. This is shown in table 1.

Element	S1	S2	S3	S4
a	0	1	0	1
b	1	0	1	1
c	0	0	1	0
d	1	0	1	1

Table 1. Characteristic matrix from the shingling

From the above table, min-hash values can be generated by picking from a permutation of the rows and for each column(shingle set) the minhash is the value of the with the first occurrence of 1. This is repeatedly done to yield a minhash signature.

- **Locality Sensitive hashing**. In this step, we set a threshold t for the Jaccard similarity as explained in section 2.1. We proceed to compare the minhash signatures of two documents and expect that their similarity exceeds the threshold t.At this point, we are using the compact minhash values as representatives of the original data.

Besides, document/text similarity, LSH is also popularly used for image similarity. With image similarity, the images are processed with global descriptors such as color histograms and then compared with distance metrics as shown in figure 5. LSH is employed by Google together with other variations to implement the VisualRank algorithm for image search [6].

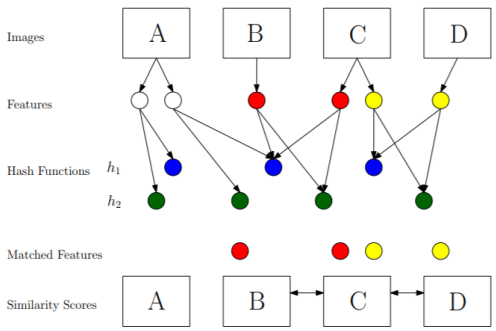


Fig. 5. Features hashed into the same bin are considered similar when using LSH for image similarity [6].

2.3 Locality Preserving Hashing

As previously discussed, LSH might have occurrences of two dissimilar data points ending up in the same hash bucket which reduces the efficiency of the hashing and also two similar data points ending up in different hash buckets which reduces the correctness (i.e. query accuracy) of a hashing method. To contrast LSH, locality preserving hashing ensures that similar data points have high probabilities to be put to the same bucket [10].

Locality Preserving Hashing is a hashing technique that preserves the neighborhood structure during quantization stage in hashing as explained in section 2.1.1. This captures the meaningful neighbors with hashing and this structure is maintained in the output as opposed to LSH that hashes randomly[16]. This emphasizes the aspect of data dependency as the similarity structure is determined by a distance metric - usually euclidean distance between the examples in given data. With the combination of these two main aspects, LPH is able to be a reliant approximation hashing technique for nearest neighbor search.

From the hashing steps explained in section 2.1.1, the neighborhood structure is preserved by combining the two step learning procedure into a joint optimization problem as proposed by Kang et al [16]. The joint optimization framework simultaneously maintains the locality preserving property in the two steps. The function is defined as[16];

$$H(Y,W) = tr\{W^T X^T LXW\} + \rho \|Y - XW\|_F^2 \quad (2)$$

where;

- X is the data $X = [x_1, x2, \cdots, x_n] \in R^{n \times d}$
- Y is the hamming representation of X calculated by $y_{ik} = sgn(w_T^k x_i + b_k)$ which gives the k^{th} hash code.
- W is the projection matrix $W = [w_1, \cdots, w_k] \in R^{d \times K}$
- ρ is the positive parameter controlling tradeoff between projection and quantization
- $X^T LX$ are the eigen vectors and eigen values and L is the Laplacian matrix given by L = D-W; D = $\sum(W)$
- $tr\{.\}$ is the trace of the matrix

By minimizing the above equation 2, the neighborhood structure is well preserved in one step and ensures that the projection matrix learned in the projection stage is not destroyed in quantization stage[16].

To inspect the locality preservation, we can observe the projection matrix $W \in R^{d \times K}$ as K hyperplanes and the projection vector p. According to Kang et al [16], $sgn(p)$ is the vertex of the hypercube $-1, 1^K$ which maps to p in terms of euclidean metric. The closer $sgn(p)$ and p are, the more the neighborhood structure is preserved as shown in figure 6. The joint optimization framework will further make W determined by both the projection and quantization stages.

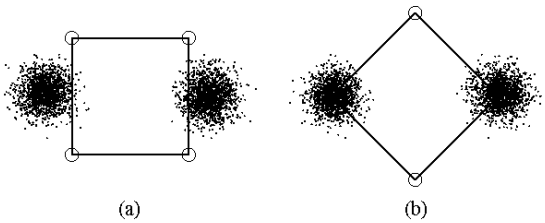


Fig. 6. Illustration of quantization stage with the circles representing the vertices. (a) shows traditional quantization without locality preservation. (b) shows neighbor preservation with vertices $sgn(p)$ and projection vectors p [16]

The locality preserving projections implemented by LPH can be witnessed in figure 7 which shows the comparison between a random projection and the locality preserving projection. It shows optimal preservation of the neighborhood structure [4].

The main advantage of Locality Preserving Hashing is it simultaneously minimizes the quantization loss, average projection distance between data points while also maintaining the neighborhood structure of the data.

3 Experiments

This section explains the experiments used to validate our investigation and reports on the results.

For this investigation, we are basing our discussion on results conducted by Yi-Hsuan Tsai and Ming-Hsuan Yang [13] which explores a comparative analysis on several hashing techniques that also include the ones explored in our investigation - Locality Sensitive Hashing and Locality Preserving Hashing.

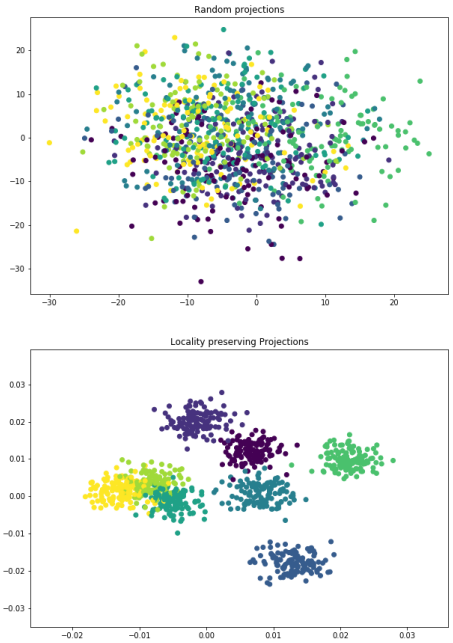


Fig. 7. Random 2D projection of a high dimensional data Vs. Locality preserving projection

The datasets explored are the CIFAR-10 and CIFAR-100 image datasets.The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. The dataset is divided into 5 training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. From the experiments carried out by Yi-Hsuan Tsai and Ming-Hsuan Yang [13], 200 images are randomly selected from each class to form a training set of 2000 images together with associated labels.

CIFAR-100 is also similar to CIFAR-10, with 100 classe with 600 images per class. Each image is 32x32 with 500 training samples and 100 test. For their experiments[13], 20 images are randomly sampled from each class creating a training set of 2000 images.
A hyper-parameter of $k = 100$ nearest neighbors is used for these experiments and similarity is based based on the hamming distance metric.

3.1 Results & Discussion

From the results, which explore multiple hashing technique implementations based on their original conceptions i.e. Spectral hashing(SH)[15], Kernelised Locality-sensitive Hashing(KLSH)[9], Binary Reconstructive Embeddings(BRE)[8], we can observe our main investigation techniques- Locality Sensitive Hashing(LSH) and Locality Preserving Hashing(LPH).

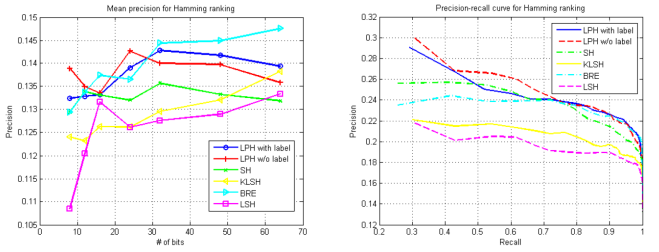


Fig. 8. (left) CIFAR-10 precision for top 500 returned samples.(right) CIFAR-10 precision-recall curve for 64 bit hash codes[13]

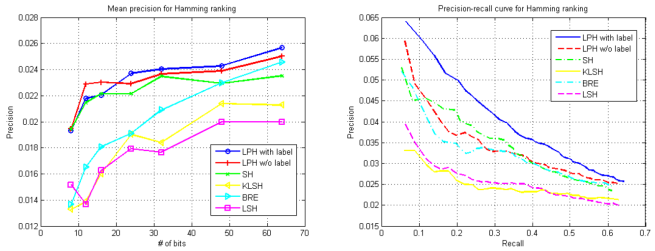


Fig. 9. (left) CIFAR-100 precision for top 50 returned samples.(right) CIFAR-100 precision-recall curve for 64 bit hash codes[13]

From both figures 8 and 9, we can observe that LSH performs worse than almost all techniques including LPH with labels and LPH without labels. Increase in the number of bits of the hash key also increases the precision of LSH but this is a general trend across all of the techniques examined. LPH with/without labels also noticeably outperforms LSH which facilitates the argument that semantic similarity from the labels of the images is much more instrumental in determining the similarity of data than the data/feature similarity.

Additionally, with the smaller and less complex dataset CIFAR-10, LPH without labels performs much better than the LPH with labels. In these experiments, the labels are used to adjust the data distribution and approximating the eigenfunctions[13]. This leaves the discussion that the labels do not contribute to LPH for simpler datasets. However, with a more complex dataset like CIFAR-100, LPH benefits from the label information as evident in figure 9. LPH thus returns more consistent results to queries than other hashing techniques for hashing-based approximate nearest neighbor search.

These experiments are evaluated based on accuracy metrics i.e. precision and recall but leave the attribute of query time unexplored which is the advantage that Locality Sensitive Hashing(LSH) has over LPH. As demonstrated above, LSH performs worse on grounds of accuracy but if the search is reliant on approximates then depending on the application, the performance can be overlooked. Theoretically, as hashing is done randomly for LSH, with a restrictive number of hashing bits, matching between a query and data samples has a faster and linear query time - $O(n^{\rho})$ and a smaller memory requirement. This is further explored by Huang et al. [5].

There is a tradeoff between query time and accuracy of approximate nearest neighbor search and this influences the use of LSH and LPH. Intuitively, accuracy might seem as an obvious choice but this is dependent on the application for which the techniques are to be used.

4 Conclusion and Future Work

In this paper we explore two hashing techniques- Locality sensitive hashing and Locality preserving hashing for approximate nearest neighbor search. We explore usage scenarios, alternative algorithm from various research publications and provide a comparative study on the two techniques with results from experiments on CIFAR-10 and CIFAR-100 image datasets. Locality Preserving Hashing has a better accuracy than Locality Sensitive Hashing. However, LSH has a faster query time than LPH. This leaves a discussion on the trade-off between accuracy and query time plus storage requirement and their influence on the application in which the techniques are to be used.

For future work, more research can be done to expand the hashing techniques to incorporate the pros from both the locality sensitive hashing and locality preserving hashing into one collective technique that is accurate, has a reasonably faster query time and less storage requirements.

Acknowledgements

The authors wish to thank the expert reviewer Kerstin Bunte for the indispensable guidance and insights into this research topic.

References

[1] A. Andoni, P. Indyk, T. Laarhoven, I. P. Razenshteyn, and L. Schmidt. Practical and optimal LSH for angular distance. *CoRR*, abs/1509.02897, 2015.

[2] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry*, SCG '04, pages 253–262. ACM, 2004.

[3] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529. Morgan Kaufmann Publishers Inc., 1999.

[4] X. He and P. Niyogi. Locality preserving projections. In *Proceedings of the 16th International Conference on Neural Information Processing Systems*, NIPS'03, pages 153–160, 2003.

[5] Q. Huang, J. Feng, Y. Zhang, Q. Fang, and W. Ng. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proc. VLDB Endow.*, 9(1):1–12, Sept. 2015.

[6] Y. Jing and S. Baluja. Visualrank: Applying pagerank to large-scale image search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30:1877–1890, 2008.

[7] W. Kong and W.-J. Li. Isotropic hashing. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, pages 1646–1654. Curran Associates Inc., 2012.

[8] B. Kulis and T. Darrell. Learning to hash with binary reconstructive embeddings. In *Proceedings of the 22Nd International Conference on Neural Information Processing Systems*, NIPS'09, pages 1042–1050, 2009.

[9] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. *2009 IEEE 12th International Conference on Computer Vision*, pages 2130–2137, 2009.

[10] M. Niu, L. Wu, and J. Zeng. Locality preserving hashing for fast image search: theory and applications. *Journal of Experimental & Theoretical Artificial Intelligence*, 29:349–359, Mar. 2017.

[11] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[12] R. Salakhutdinov and G. Hinton. Semantic hashing. *Int. J. Approx. Reasoning*, 50(7):969–978, July 2009.

[13] Y. Tsai and M. Yang. Locality preserving hashing. In *2014 IEEE International Conference on Image Processing, ICIP 2014, Paris, France, October 27-30, 2014*, pages 2988–2992. IEEE, 2014.

[14] J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. *CoRR*, abs/1408.2927, 2014.

[15] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *Advances in Neural Information Processing Systems 21*, pages 1753–1760. Curran Associates, Inc., 2009.

[16] K. Zhao, H. Lu, and J. Mei. Locality preserving hashing. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 2874–2880. AAAI Press, 2014.

The application of machine learning techniques towards the detection of fractures in CT-scans of the cervical spine

Kareem Al-Saudi and Frank te Nijenhuis

Abstract— Deep learning methods are increasingly being applied within the context of medical imaging with great success. One area of medical imaging that's yet to be explored is the use of such methods and techniques to automate the detection of fractures of the cervical spine on a CT scan. We survey relevant existing literature, examining effective methods of fracture recognition as well as many of the important issues that are frequently encountered when considering the approach of deep learning in the case of medical images. Finally, we conclude by proposing an approach to solving this problem using a variation of the Inception v3 architecture.

Index Terms—Deep learning, radiology, image classification, image segmentation, medical imaging

1 Introduction

Approximately 7% of trauma patients entering a hospital's emergency department (ED) after a motor accident have sustained some type of injury of the cervical spine (neck) [8]. After such an accident, the neck of a patient being transported to the hospital has to be fixated to prevent any movement, which could significantly worsen the outcome in case of a spinal fracture, as the bone fragments might slide around to compress the spinal cord (*myelum*). This has to be done even if there is no evidence of neurological deficits at the time of the accident. Often, to rule out neck fractures, a radiograph or, in certain cases where the radiograph does not provide satisfactory results, a CT-scan of the cervical spine is made.

An on-call radiologist evaluates the scan to determine whether or not there is a fracture. To the radiologist, reading the scan is a time-consuming process, and there might be additional delay because he or she is not always available to look at the scan right away. Automating the process of fracture detection in the cervical spine is therefore desirable. An algorithm for automatically triaging the scans based on severity would speed up the process of radiological evaluation. In this setup, if a scan is triaged as high priority, it has to be seen by the radiologist immediately. If, however, no fracture is detected by such an algorithm, the radiological evaluation can wait. Similar schemes have already been succesfully applied in the detection of chest pathology on radiographs [2]. Evaluating such a scan requires an efficient and robust object recognition algorithm. Currently, the most effective solutions to these kinds of object recognition problems are Convolutional Neural Networks.

In this paper, different techniques and methodologies within the field of medical imaging will be discussed. We will summarize the different approaches used by researchers in the past and in doing so will be able to confidently select the appropriate approach with regards to detecting fractures in CT-scans of the cervical spine.

The remainder of the paper will be organized as follows: the remainder of section 1 serves to briefly discuss the concept of neural networks and machine learning as well as to introduce the anatomy of the cervical spine and some of the basic ideas behind CT-scanning. Section 2 explains Computed Tomography (CT) in detail. Section 3 concisely describes the methodology that we have utilized to arrive at our conclusion. Section 4, the discussion, is where the bulk of our work lies. This section discusses the different approaches and results from a variety of papers within the field of machine learning in medical imaging and their relevance to our overall approach. Finally, section 5 gives a brief summary of the content of this paper and eventually concludes with our own theoretical solution to the problem of detecting fractures in CT-scans of the cervical spine.

- *Kareem Al-Saudi is a CS Master Student, E-mail: k.al-saudi@student.rug.nl.*
- *Frank te Nijenhuis is a CS Master Student, E-mail: f.g.te.nijenhuis@student.rug.nl*

Fig. 1. Image showing a slice of a CT-scan of the cervical spine. Sagittal slice in bone window setting. Image courtesy of Dr Bruno Di Muzio, Radiopaedia.org, rID: 39801

1.1 Neural networks and machine learning

Machine learning can be seen as a subfield of the more general scientific field of Artificial Intelligence (AI). In a similar manner, deep learning is a subfield of machine learning. In this classification, the 'deep' term comes from the fact that neural networks with multiple hidden layers are being used, as opposed to the shallow neural networks used in the early days of machine learning. In general, the most effective image classification methods currently being applied in the field of medical image recognition are Convolutional Neural Networks (CNN), an example of a deep learning technique. As such, we mainly focus our analysis on these techniques as well, our analysis is restricted to deep learning techniques in medical imaging. To introduce CNN we should first consider the more general versions, Feedforward Neural Networks (FNN) and Artifical Neural Networks (ANN) [12, 14]. An ANN is a computational processing system that is loosely inspired by the way biological nervous systems, such as the human brain, operate. It is comprised of interconnected computational nodes, referred to as neurons. The amount of layers and nodes per layer is determined by the so-called hyperparameters of the model. The hyperparameters are set at the beginning of the training phase of the model. This in contrast to the parameters of the model, which change during training. There are many different types of ANN. One of these types is the FNN, upon which an additional constraint has been imposed, namely that the information being processed by the system can only flow from the input

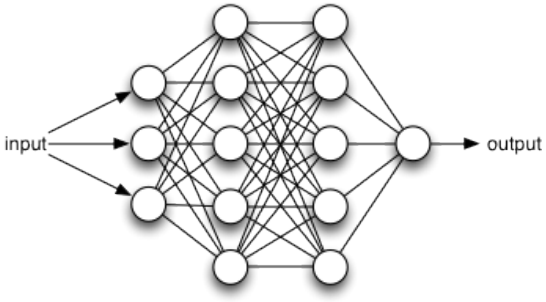


Fig. 2. A simple three-layered 'feedforward' neural network (FNN). It comprises of three layers: an input layer, a hidden layer and an output layer. This structure is the basis of which most common ANN architectures are built.

towards the output. A network upon which this constraint is not imposed is called a Recurrent Neural Network (RNN), since information can 'recur' from a deeper layer back to an earlier layer. Recurrent Convolutional Neural Networks have been described in literature [11], but they will not be discussed in this paper as they have not been applied to medical imaging. Mathematically speaking, a FNN can be seen as a directed acyclic graph, whereas a RNN is a directed graph, since cycles are possible in this case. Through a series of self-optimizations the FNN is capable of training itself to produce a certain desired output based on an input vector, a process called 'learning'. As such, FNN are used in supervised learning models, where labeled data is available so that the correct output is known given each input in the training set.

The process can be described as follows. Input values are loaded, usually in the form of a vector, to the first network layer, aptly named the input layer. Each layer contains multiple neurons, and each neuron computes a weighted sum of its inputs. The result passes through a nonlinearity after which it is propagated to the neurons in the next layer. In this context, 'learning' refers to adjusting the weights between the neurons to nudge the system towards a correct output for a particular input. This learning is directed by the minimization of a cost function. Having multiple hidden layers of neurons stacked upon each other is what is commonly referred to as deep learning. Figure 3 illustrates the basic structure of a feedforward neural network.

CNNs are analogous to traditional FNNs with the main difference being that they have multiple specialized network layers, which are particularly suitable for use in image recognition. The first part of the CNN is an optional preprocessing layer, with fixed filters which are not changed during the learning process. This can be interesting if we want to artificially force the network to focus on particular features. After the image processing layer there are multiple convolutional layers, from which the network derives its name. A convolutional layer works by convolving the input image with a certain kernel. The convolution can be seen as a filter sliding over the input image, looking for certain low level structures such as edges. The kernel types are fixed, they are not adjusted through training. The input image is usually convolved with multiple different types of kernels. Each neuron in the layer receives information from a small input area of the input image called the receptive field. The result of this operation flows to the next layer of the network.

Intermixed with the convolutional layers, there are pooling layers which combine the information from the previous layer. Commonly used types are either max-pooling layers or averaging layers. Both types of layers again have a receptive field which is larger than their output size, therefore they subsample their input. A max-pooling layer propagates the maximum value in its input field, whereas the averaging layer propagates the average value. Many more different ways of handling this subsampling have been proposed.

After multiple convolutional and pooling layers, there are fully connected (FC) layers, which are similar in structure to the layers of a regular ANN as described earlier. These classification layers combine the outputs of the convolutional and pooling layers to draw conclusions about higher level concepts. By the time information arrives at the FC layers, it is no longer influenced by the spatial relations in the input, instead relying on the actiations of the pooling layers to determine whether or not a feature is present in the image as a whole. In this way, information is abstracted from the image in a space-invariant manner. This is inspired by the functional organization of the primary visual cortex in the primate brain. Through this simple and elegant idea, abstract visual concepts can be extracted from images [6].

The Inception architecture created by Google is frequently used in medical imgae recognition because it performs well on many imaging tasks while at the same time being computationally inexpensive when compared to other leading architectures [20]. Another architecture which is frequently used in modern image recognition is U-Net, so-called because of the characteristic U shape when drawing out its layers as a graph. U-net is particularly capable of handling small image sets by effectively exploiting the infromation gained from data augmentation [16].

Machine learning techniques such as neural networks can be used for the classification of an image, where the output of the network is a list of probabilities for each of the different possible output classes, but we might also want to perform segmentation. In segmentation, the machine learning system also tells us where the lesion is located. In medical imaging, segmentation is particularly useful as a medical professional is usually not merely interested in whether there is an abnormality in an image, he also wants to know where it is located.

In addition, machine learning techniques can also be applied to other parts of medical imaging, such as in the denoising of low dose CT scans [4]. A low-dose CT scan is often noisy as a result of the low radiation dose not penetrating all tissues adequately. Denoising CNN can be used to reconstruct higher resolution CT's. We will not consider denoising techniques in this text.

1.2 The cervical spine

The cervical spine consists of 7 vertebrae, as depicted in Figure 4. It runs from the base of the skull to the opening of the thorax (ribcage), at the first rib. It is more mobile compared to the lower, sturdier, levels of the spine. The cervical spine acts as a fulcrum on which the head, which is relatively heavy in humans, pivots. The 7th vertebra (*vertebra prominens*) of the cervical spine can be easily identified as it protrudes prominently from the back of the neck. The first 2 vertebrae have specific names, the *atlas* and *axis*, respectively. They are special in structure and function, with the *atlas* acting as a platform to stabilize the skull and the *axis* facilitating rotation of the skull through a tooth like protrusion called the *dens*, which interlocks with an opening in the *atlas*. The cervical spine is held together by a complex system of ligaments.

There are multiple different typical fractures in the neck. This is because certain parts of the cervical spine are predisposed to break more easily than other parts. Most fractures occur at two levels, approximately half of the fractures occur at the level of C6 or C7, and approximately one third occur at the level of C2. Usually, the mechanism of injury is hyperflexion, where the neck bends forwards further than it is supposed to, and the ligaments at the back are torn. The main danger is always the same, compression of the spinal cord, which might lead to neurological deficits or even tetraplegia, the loss of sensory and motor function in all limbs.

1.3 CT

Computed Tomography (CT) is an imaging technique where multiple X-ray measurements are combined to obtain a single stack of image slices (cf. Greek $\tau o \mu o \sigma$ tomos, "slice" and $\gamma \rho \alpha \phi \omega$ graph, "to write"). The CT-scan is made using a scanner apparatus, a large torus shaped device into which the patient can be inserted. Inside the scanner, a construction containing an X-ray emitter and an opposing detector is rotated around the patient. From the measurements obtained at multiple different rotational angles, stacks of image slices through the human body can be algorithmically reconstructed. In the hospital setting, CT-scans have become an invaluable tool, not just in the diagno-

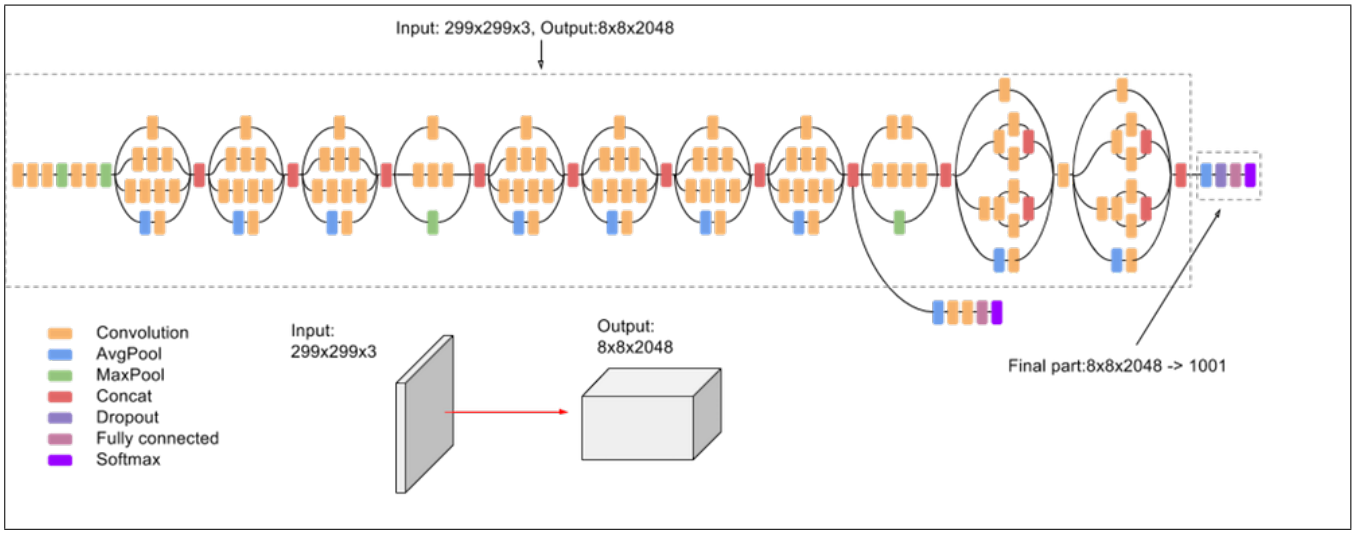


Fig. 3. The Inception v3 model is made up of symmetric and assymetric building blocks. As seen above, this includes convolutions, average pooling, max pooling, concats, dropouts and fully connected layers.

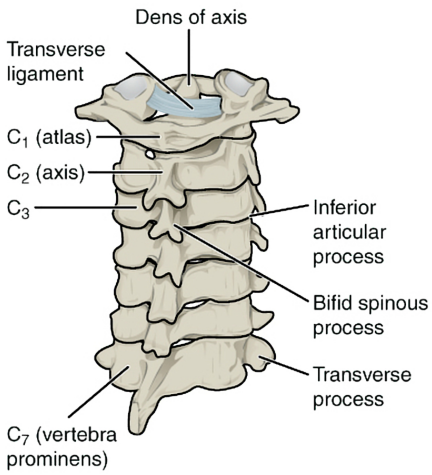


Fig. 4. Schematic representation of the cervical spine, showing the 7 vertebrae commonly abbreviated as C1-7. In particular, note the special shape of C1 (atlas) and C2 (axis). Image courtesy of OpenStax College, Radiopaedia.org, rID: 42770

sis of disease, but also in preoperative planning and in the follow-up of certain diseases. Many different types of CT-examinations have been developed for specific purposes, such as CT Angiography (CTA) to visualize blood vessels, and dual energy CT to investigate for instance liver lesions (a lesion is any abnormality in tissue structure, such as a tumor). For more information about CT-scans, see for instance [18]. After processing, the acquired CT data is stored in DICOM format, which is a standardized imaging format for medical images. The density of a voxel on a CT-scan can be expressed in Hounsfield units, which is a way of measuring the signal attenuation. Air has a value of -1000 HU, and water has a value of 0 HU. Metals and other foreign objects give the highest density on a CT-scan, which is around 30000 HU. A CT-scan which has been stored like this has a high dynamic range, it contains more information than can be shown using a default 256 value grayscale colormap. To avoid high contrast issues, radiologists use windowing settings when looking at the scans. The windowing is defined by its level, and its width, where the level indicates the midpoint of the window in Hounsfield Units, and the width determines the upper and lower threshold of the values displayed on the CT. In this way, the extra information from the high dynamic range can be mapped to grayscale images. A narrow window can be used to examine tissues exhibiting minor signal changes, such as the soft tissues in the abdomen. Usually, the radiologist examines the same scan in multiple different windowing settings to look for different abnormalities. Because of the way the detector moves around the patient, the CT-scan consists of a stack of axial slices. The axial direction is one of the cardinal anatomic planes of the body, it intersects the body horizontally. When examining the scan, a radiologist may want to view different planes, which can be reconstructed from the axial stack using software. The two other planes are known as the sagittal plane (cf. latin *sagittarius*, archer), which intersects the body from the side (think of the pose of an archer) and the frontal plane, which represents a vertical section through the human body.

2 Methodology

We review the available literature on the topic of image recognition in radiology, so that we may distill an effective method to detect fractures of the cervical spine on CT images. The eventual goal is for this proposed method to be applied in clinical practice, where it could be used to determine whether a CT-scan of the cervical spine shows signs of serious injury. If this is the case, the radiologist will have to look at it as soon as possible. If the system determines that no spinal injury is present, the scan is put on a low priority list to be reviewed later by the radiologist. By creating such a method we hope to streamline the workflow of the radiologist. Based on this task description, we are interested in a method which not only accurately diagnoses fractures, but which also has an extremely low rate of false negatives, as a false negative result potentially means a delay in diagnosis. Note that in this setup, every scan is still reviewed by the radiologist at some point.

We would also like to preface the discussion by stating that the techniques and methods that are about to be discussed, while not entirely relevant to our own problem, should serve as a solid base of knowledge and understanding that will be the basis of our theoretical approach's composition. This is because, while the datasets and approaches might not be the same, the techniques and methodologies applied in training neural networks usually still apply regardless of where they lie on the medical imaging spectrum.

3 Discussion

We consider the problem of automatically recognizing fractures on a CT-scan using machine learning techniques. To formulate an optimal approach to solve this complex problem, we break it down into different subproblems, and we summarize what is known about solving each subproblem. We conclude by proposing a method which we think would perform best by combining the best solutions to each of the subproblems.

The majority of AI papers in radiology are focused on radiograph imaging. This is a different imaging paradigm from the CT scans we are interested in. A radiograph is a 2D greyscale image of a part of the body, whereas a CT-scan is a 3 dimensional examination consisting of a stack of 2D images. We have still included many results from papers focusing on radiographs because there a sparsity of literature on the subject of machine learning on CT. Typically, a radiograph has a higher resolution (usually, a radiograph contains 2048×2048 pixels) than a single slice of the CT scan (usually, 512×512 pixels), this is an important detail to keep in mind during our analysis.

3.1 Dataset

A common problem with the application of machine learning techniques in medical imaging is the low availability of data. Medical data is confidential, and it is relatively hard to obtain permission to create datasets of the size necessary for the effective application of machine learning algorithms. First, lists of patients who have had a scan of the cervical spine have to be obtained from the hospital database. Next, these patient lists have to be annotated and combined into a proper dataset. For instance, since not everyone who has had a CT-scan of the cervical spine actually broke their neck, we need to annotate those scans where a fracture is present. Since we also require some kind of ground truth, we need a radiologist to indicate where the fracture is present within the scan, using some kind of annotation pen tool. In general, luckily, only a small percentage of patients scanned will actually have a spinal fracture. This does, however, exacerbate the issue of small datasets, because in the dataset which is already small, we have a relatively low percentage of informative scans. Multiple solutions have been formulated to solve this problem. The easiest method would be to manually rebalance the dataset by changing the ratio of scans containing a fracture to scans which do not contain a fracture.

In Lindsey et al. [13], an initial bootstrapping period was used. The goal here was to accurately detect fractures on wrist radiographs using a CNN. During bootstrapping, multiple types of x-ray images were fed to the network, such as x-ray images of the chest and knee. This was done to teach the network how to read an x-ray image in general, after which it can be trained specifically for the detection of wrist radiographs. This somewhat alleviates the problem of having a small dataset, as initial training can be performed on a different, larger but less specific set of images. Data was further augmented utilizing various techniques such as rotating, cropping and contrast variation. This is done to make the model more robust to rotational, translational and contrast changes and to allow for a larger dataset with which to train the neural network. Instead of training our network with CT-scans of the cervical spine immediately, we could first initialize it with scans from different parts of the body, the idea being that it first learns what to look for in a scan in general before progressing to the more specific case of cervical scans. In CT-scans, this technique only makes sense for the detection of bone abnormalities, however, as for instance a CT-scan of the abdomen contains tissues of an entirely different structure. Still, obtaining such a dataset might prove useful in the initial training sessions of the network.

Another way to deal with the relative lack of data in medical imaging as opposed to other fields is to simply train shallower networks [3]. The idea behind this approach is that a shallower network contains less parameters, and therefore requires less training data to be successful. A shallow network runs a higher risk of misclassifying some of the harder scans, so care should be taken when using this technique. Techniques for finding the optimal architecture exist, these might prove helpful when designing the initial architecture. This is called hyperparameter optimization, see for instance [7, 19] for more details.

3.2 Preprocessing

Here, we consider the problem of a dataset containing extra information or noise, as is often the case when working with real medical images. For instance, textual annotations such as patient data and the date of the examination might be present on the scans. In Annarumma et al. [2], preprocessing had to be performed on the radiographs to remove such annotations. This removal was done using yet another neural network, Tensorbox, which was trained to detect artifacts. If an artifact, such as a string of text on the image, is found, it is covered with a black box. To prevent the introduction of a learning bias as a result of this black box treatment, black boxes were positioned randomly around the corners of all training images.

As mentioned earlier, another issue is the high dynamic range of a CT-scan. Since most learning architectures are not capable of dealing with the high dynamic range information from the CT-scan directly, some preprocessing step is required. Chilamkurthy et al. [5] approach the high dynamic range problem by splitting each slice into three channels based on different windowing setting. They can then feed their CNN algorithm information about the same pixel in bone setting, brain window and subdural windowing. When looking for cervical fractures, a similar approach might be necessary, where the algorithm looks at both a bone window setting and some sort of soft tissue window.

In many real worlds applications, preprocessing like this may be necessary to ensure correct results from the machine learning method being applied. In general, ensuring that the input is as standardized as possible might be desirable. In CT-scans, the same windowing setting should be used as much as possible. No standard preprocessing pipeline can be defined, as the required amount of preprocessing completely depends on the dataset being used.

3.3 Machine learning in a CT-scan

Applying machine learning techniques specifically to CT-scans brings its own suite of problems, as opposed to for instance the detection of lesions on regular radiographs or MRI's.

Deep learning in 3 dimensions, in the CT-scan, is significantly harder than learning in 2D, which is the case with radiographs, because of added computational complexity when generalizing from the usual 2D to 3D convolutions. Even though it is more difficult, Roth et al. [17] show that it is feasible to apply 3D convolutions for the segmentation of organs on abdominal CT-scans using modern hardware. Another recent study proposes a different solution, by implementing three planar CNNs in a triplanar fashion, such that axial, frontal and sagittal slices of the image are simultaneously being analyzed [15]. This is effective in the segmentation of cartilage on MRI.

The approach used in this study tackles the problem of producing false-positives over the multitude of slices in each image. This is done by allowing each CNN to dissect a different plane of said image and allowing them each to only interact between the output of their respective final layers. The outcome is fascinating; exhibiting marginally higher degrees of accuracy, sensitivity and specificity while demonstrating the ability to procure results 10-15 times faster than the 3D CNN. Furthermore, their method allows for the automatic segmentation of a 3D image allowing it to be fed slice-by-slice to the 2D CNNs.

One approach utilized by Chilamkurthy et al. [5] is to identify intracranial bleeds by running a version of the ResNet18 architecture with 5 fully connected layers on each individual slice, and then combining the confidence levels on each slice using a random forest to say something about the entire scan.

Kwan et al. [10] demonstrate the use of histograms for identifying structures of interest in CT scans of the spine. This method can be used to first extract the vertebrae contours from the scan. Automatic histogram thresholding is performed to obtain the region of interest. Noise and artifact removal are also performed using a detection algorithm and the morphological erosion operator, respectively. Finally, a 2 stage boundary rectification algorithm is applied to end up with

a reliable vertebral contour. Afterwards, CNNs could be used for the classification, even though this is not done in the original paper or elsewhere to the best of the authors' knowledge. It might be interesting to look into the combination of deep learning with morphological processing.

3.4 Learning architecture

Lindsey et al. [13] provide a neural network to assist Emergency Department (ED) physicians with a tool to detect fractures in wrist radiographs. Specifically, they use an extension of U-Net. Subspecialized radiologists are asked to draw a bounding box around the fractures they find using a special tool. The neural network is then trained to emulate this bounding box classification approach, leveraging the expertise of the radiologists. The output of the CNN is a single fracture probability combined with a heat map, which can be overlaid on the image to show the probable location of the fracture.

One of the limitations of this study is that the diagnostic value is limited because of the retrospective nature of the research. The experts do not actually see the patient, they just interpret the scans. This means that valuable clinical information gained through talking to the patient and from further physical examination is missing.

In Annarumma et al. [2], a study investigating automated triaging of chest radiographs, researchers trained an ensemble of 2 CNNs to classify chest radiographs as either critical, urgent, nonurgent, or normal. This classification is then used to prioritize the images to reduce the average waiting time for critical images.

These CNNs both used the Inception v3 architecture but were trained on completely different datasets. The first network is applied to a downsampled version of the images at a resolution of 299x299 pixels. The reasoning behind this is that while detailed information is lost the network becomes proficient at detecting global abnormalities. The second network, on the other hand, operates on higher resolution images rendering it better suited for detecting visual patterns that may otherwise be missed at a lower resolution.

Interestingly, using two CNNs leads to some riveting conclusions. It was found that averaging the results of these two subsampled networks worked better than using a single higher resolution architecture. ADADELTA was used for optimization, which is a novel per-dimension learning rate method for gradient descent boasting trivial computational overhead compared to other similar methods rendering it suitable to be applied in a variety of different situations. The reason for its success lies in its ability to automate the tuning process of determining the learning rate which is, in other cases, chosen by hand. The importance in this lies within a rate that is both high enough to ensure fast learning but not too high so as to cause the system to diverge in terms of the objective function [21]. Training was done by starting 68.000 iterations from random weights. The model with the best average F_1 score was selected for testing.

The difference between our proposed setup and this paper is that this network also gives the probability of the specific type of abnormality using ordinal regression. This is not something we plan to do in the current setup. Rarer conditions are included in the dataset to ensure that the system can also draw conclusions about this. This is something which has to be accounted for in the training dataset composition.

In Kamnitas et al. [9], a similar parallel scheme is used. 2 CNNs receive input from the same image location. One CNN has a receptive field which looks at the neighborhood of the voxel, the other receives a subsampled version which looks at a broader 'context' of the same voxel. In doing so, the prediction accuracy increases because of the addition of this contextual information to the next layers.

3.5 Statistical analysis

To see whether the chosen learning architecture is adequate we need to analyze its performance. Many different performance measures have been proposed for usage with neural networks. In our case, the network classifies a scan based on whether it contains a fracture or not. We will therefore consider classification based performance measures. With our particular setup, we need a method that minimizes the amount of false negatives, as this will result in fractures being missed which is the main priority. We will minimize the number of false negatives at the expense of an increase in the number of false positives as missing a fracture poses an extreme risk to the patient.

This approach works in our favor as, within the context of machine learning in medical imaging, neural networks were never meant to replace the radiologist but rather serve as a means to simplify and otherwise assist with their work. Radiologists will not be replaced by machines; rather, "radiologists of the future will be essentially medical data scientists ." [1].

To quantify accuracy, we may calculate the F_1 or Dice coefficient, which is expressed as

$$F_1 = 2\frac{P \cdot R}{P + R}$$

where P indicates the positive predictive value, which is the proportion of actually positive results within the set of positively predicted results, in our case the number of actual fractures detected by the system divided by the total amount of predicted fractures. R indicates the recall or sensitivity, which is the amount of actually positive results detected by the system divided by the total number of positive results, so including the fractures missed by the system. The F_1 score obtained in this way ranges between a value of 1, perfect detection of all fractures but no excess detection, and 0.

4 Conclusion

To summarize, in this paper, we have presented a brief introduction as to why automating the detection of fractures in CT-scans of the cervical spine would be extremely beneficial to radiologists. We have done this by first providing a brief overview of neural networks and machine learning, a succint summary of the anatomy of cervical spine, a concise explanation on computed tomography and its relevance to our paper. After this introduction we discussed the importance of many factors influencing the final performance of the algorithm, such as the dataset, preprocessing steps, and learning architecture along with machine learning techniques in a CT-scan as well as a brief statistical analysis.

Given our study of all of this pre-existing information over numerous prior architectures and their application in the field of medicine we can start theorizing as to which methodology best suits our problem. To overcome issues posed by a small dataset, which is a realistic scenario as collecting enough data to properly train a CNN can prove quite difficult in medicine, we might try to begin training on a larger, less specific dataset and then later focus on images of the cervical spine. We may also choose to augment our dataset to produce a larger dataset out of our existing data. Preprocessing the scans might be necessary as artifacts may be present in our CT-scans. This can be done by using tools such as Tensorbox which have proven to be efficient for this purpose. To determine an optimal learning rate, the ADADELTA algorithm can be used. Because the datapoints in our dataset are 3-dimensional scans, we know that a 3D CNN would be the most natural choice. However given the memory requirements of fully 3D networks as well as previous discussions proving that a series of three orthogonal 2D patches can be just as, if not more effective in producing results with high degrees of accuracy we can conclude that a triplanar 2D CNN is most suitable in our case. Naturally, as a result of this, we will train a CNN receiving information from 3 2D convolutional patches to recognize fractures of the cervical spine. It seems that a modified version of the Inception v3 architecture to facilitate triplanar learning would be an adequate starting point for the training process. Hyperparameter-optimization techniques can be employed to further refine the most suitable architecture, and experimentation with different architectures such as U-Net might provide an effective alternative to Inception. Finally, to produce even better results we will combine the output of two networks, one of which receives a subsampled version of the input, to improve the overall accuracy of the system. To evaluate the final performance of the algorithm, a simple Dice coefficient can be used.

ACKNOWLEDGEMENTS

The authors wish to thank Michael Biehl who has reviewed this paper as an expert within the Machine Learning and Computational Intelligence field. In addition, the authors also want to thank our colleagues Codruț-Andrei Diaconu and Sofie Lovdal who have reviewed this paper as well. The authors would also like to give specials thanks to Rein Smedinga, and Femke Kramer who provided useful guidance along the way.

REFERENCES

[1] Artificial intelligence in radiology: The game-changer on everyone's mind, Jan 2019.

[2] M. Annarumma, S. J. Withey, R. J. Bakewell, E. Pesce, V. Goh, and G. Montana. Automated triaging of adult chest radiographs with deep artificial neural networks. *Radiology*, 0(0):180921, 0. PMID: 30667333.

[3] J. R. Burt, N. Torosdagli, N. Khosravan, H. RaviPrakash, A. Mortazi, F. Tissavirasingham, S. Hussein, and U. Bagci. Deep learning beyond cats and dogs: recent advances in diagnosing breast cancer with deep neural networks. *The British Journal of Radiology*, 91(1089):20170545, 2018. PMID: 29565644.

[4] H. Chen, Y. Zhang, W. Zhang, P. Liao, K. Li, J. Zhou, and G. Wang. Low-dose CT via convolutional neural network. *Biomed Opt Express*, 8(2):679–694, Feb 2017.

[5] S. Chilamkurthy, R. Ghosh, S. Tanamala, M. Biviji, N. G. Campeau, V. K. Venugopal, V. Mahajan, P. Rao, and P. Warier. Deep learning algorithms for detection of critical findings in head ct scans: a retrospective study. *The Lancet*, 392(10162):2388–2396, Dec 2018.

[6] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, IJCAI'11, pages 1237–1242. AAAI Press, 2011.

[7] M. Claesen and B. D. Moor. Hyperparameter search in machine learning. *CoRR*, abs/1502.02127, 2015.

[8] A. Kaji and P. S. Hockberger. Spinal column injuries in adults: Definitions, mechanisms, and radiographs (uptodate), 2019.

[9] K. Kamnitsas, C. Ledig, V. F. Newcombe, J. P. Simpson, A. D. Kane, D. K. Menon, D. Rueckert, and B. Glocker. Efficient multi-scale 3d cnn with fully connected crf for accurate brain lesion segmentation. *Medical Image Analysis*, 36:61 – 78, 2017.

[10] F. Kwan, I. Gibson, and K. Cheung. Automatic boundary extraction and rectification of bony tissue in ct images using artificial intelligence techniques. 2000.

[11] S. Lai, L. Xu, K. Liu, and J. Zhao. Recurrent convolutional neural networks for text classification, 2015.

[12] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

[13] R. Lindsey, A. Daluiski, S. Chopra, A. Lachapelle, M. Mozer, S. Sicular, D. Hanel, M. Gardner, A. Gupta, R. Hotchkiss, and H. Potter. Deep neural network improves fracture detection by clinicians. *Proceedings of the National Academy of Sciences*, 115(45):11591–11596, 2018.

[14] K. O'Shea and R. Nash. An introduction to convolutional neural networks. *ArXiv e-prints*, 11 2015.

[15] A. Prasoon, K. Petersen, C. Igel, F. Lauze, E. Dam, and M. Nielsen. Deep feature learning for knee cartilage segmentation using a triplanar convolutional neural network. In K. Mori, I. Sakuma, Y. Sato, C. Barillot, and N. Navab, editors, *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2013*, pages 246–253, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[16] O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[17] H. R. Roth, H. Oda, X. Zhou, N. Shimizu, Y. Yang, Y. Hayashi, M. Oda, M. Fujiwara, K. Misawa, and K. Mori. An application of cascaded 3d fully convolutional networks for medical image segmentation. *CoRR*, abs/1803.05431, 2018.

[18] E. Seeram. *Computed Tomography, Physical Principles, Clinical Applications, and Quality Control*. Saunders, 2015.

[19] H. Shouno and M. Okada. A hyper-parameter inference for radon transformed image reconstruction using bayesian inference. In F. Wang, P. Yan, K. Suzuki, and D. Shen, editors, *Machine Learning in Medical Imaging*, pages 26–33, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[20] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.

[21] M. D. Zeiler. ADADELTA: an adaptive learning rate method. *CoRR*, abs/1212.5701, 2012.

An Overview of Runtime Verification in Various Applications

Neha Rajendra Bari Tamboli(S3701417), Vigneshwari Sankar(S3874222)

Abstract— This paper investigates about what runtime verification is, how it is used in financial and medical application and how it aids the system by dynamic and real time analysis of data for critical decision making. Software correctness is an activity which determine whether the program satisfy or violate the given properties . A property is a set of rules to be followed. The more accurate the required results are the more correct the software is. There are verification techniques used to check the correctness. One of them is static verification analysis in which the code is built to ensure that standard coding practices have been adhered. However, there are a few major concerns with static verification techniques like, the tester may not have access to the code to check if the coding standards are followed. This is when runtime verification comes into picture. We evaluate runtime verification in medical and financial aspects and how it is a better verification approach compared to static or semi static approaches. Finally the paper discusses the observed commonalities between these two application domains, the benefits and the challenges faced.

Index Terms—Decision making, software correctness,realtime,runtime verification.

1 Introduction

Due to the large amount of data we have in today's world, it has become increasingly important to handle this data not only efficiently but also on a real time basis especially in applications where decision making is critical. One of the solutions to this was introduction of runtime verification: how data is analysed and checked on a real time basis to look for any discrepancies. To implement such a functionality, changes to existing software is made and runtime verification is added as a module to the software. A Software is a set of programs instructing a computer to do specific tasks. Software and Software systems have become pervasive in the digital world. From traditional software applications like media player, word processor etc., software plays a significant part in our everyday life. For example, to help the air traffic control system improve the safety and efficiency of the aircraft, a software called Future Air Traffic Management Concepts Evaluation Tool (FACET) was developed by NASA.[6]

In recent years, software has added a new paradigm to the architecture of software systems. They are seen more and more as autonomous agents acting according to certain properties. For such systems, verification is challenging as the overall behaviour of systems depends heavily on the agents involved. One important aspect of verification is to check whether each party acts according to the property they have agreed on.

"Instead of debugging a program, one should prove that it meets its specifications, and this proof should be checked by a computer"(John McCarthy, 1961)[12] Traditionally, *theorem proving[5] or Model Checking[13]* are the main traditional verification techniques used.

Theorem proving is fully automated which checks the correctness of the program mathematically with respect to mathematical formal specification.It is used in the hardware industry because the important aspects of hardware design are manageable to automated proof methods, making theorem proving verification easier to introduce and more productive.

In Model Checking it is checked whether the given model(hardware or software) meets the given specification such as absence of deadlock(A process in which two different systems try to access the same data at the same time)[7] that can cause the model to crash. To solve this the model of the system and the specification are formulated in some mathematical languages.

Though the formal verification techniques are effective and less complex, it is also time consuming and expensive. A non technical person will find it difficult to use this model and extensive training is required.

- *Vigneshwari Sankar is a MSc Computing Science student at the University of Groningen , E-mail: v.sankar@student.rug.nl.*
- *Neha Rajendra Bari Tamboli is a MSc Computing Science student at the University of Groningen, E-mail: n.r.bari.tamboli@student.rug.nl*

2 Runtime Verification

Runtime verification also referred to as runtime monitoring, trace analysis, dynamic analysis etc., is a lightweight, yet rigorous, formal method that complements the classical verification techniques with a more practical approach. The correctness is analyzed by means of properties. A property is a set of instructions which define the conditions that needs to be satisfied in order to ensure smooth running of the system under consideration. When a property is not satisfied it is referred to as a violation, and when these properties are analyzed real time that is when runtime verification comes into picture. It deals with inaccurate information on the behaviour of the software which is typically referred to as a violation and when such behaviour is detected an alert is sent. There are methods to identify the correctness of the execution trace currently taken into consideration. These are discussed as below.

2.1 Monitors

A monitor is merely a device that checks the correctness of the current execution trace[11]. It decides whether the current execution trace satisfies the property by showing an output of either *true/yes or false/no*[11]. The monitor can be used in 2 ways, one is when the current execution trace is taken into consideration and the other when a finite set of sequences is fed to the monitor, in the later case it is referred to as offline monitoring.

2.2 Monitoring Setup

A typical runtime verification monitoring setup consists of three main components, namely the system under scrutiny, the monitor and the instrumentation mechanism.

Monitors are computational entities that execute along side a system so as to *observe* its runtime behaviour and possibly determine whether a property is satisfied or violated from the exhibited execution. When sufficient system behaviour is observed, a monitor may reach a *verdict* i.e., either acceptance or rejection. This verdict is normally assumed to be *definite* i.e., it cannot be revised and is typically communicated to some higher-level entity responsible for handling monitor detection.

Instrumentation is the computational plumbing that connects the execution of a system under scrutiny with the analysis performed by the monitor. It typically concerns itself with two aspects of the monitoring process. First, instrumentation determines what aspects of the system execution are made visible to the monitor for analysis. Instrumentation records the relevant information from the computation of a running system and records them as system *events*. An event is a process currently taken into consideration. The recorded events are then reported to the monitor in the form of an ordered stream called an *execution trace* (of events), which would normally correspond to the same notion introduced in the previous section.

Second, instrumentation also dictates how the system and the monitor execute in relation to one another in a monitoring setup. For instance, instrumentation may require the system to terminate executing before the monitor starts running, or interleave the respective executions of the system and the monitor that share a common execution thread. In concurrent settings, the system and monitor typically have their own execution threads, and instrumentation dictates how tightly coupled these executions need to be. The instrumentation either requires the respective threads to execute synchronously which are regulated by a global clock, or else allow threads to execute asynchronously to one another and then specify synchronization points between the respective executions.

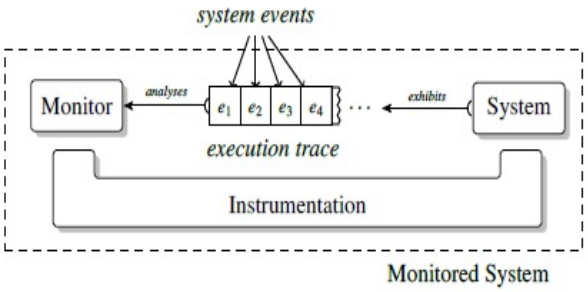


Fig. 1. The basic set up of a monitor and how it works[3]

2.3 Monitor-based Runtime Reflection

Monitor-based runtime reflection or runtime reflection (RR) is an architecture pattern for the development of reliable systems in various applications The basic architecture of the system is represented as below in figure 2.[4]

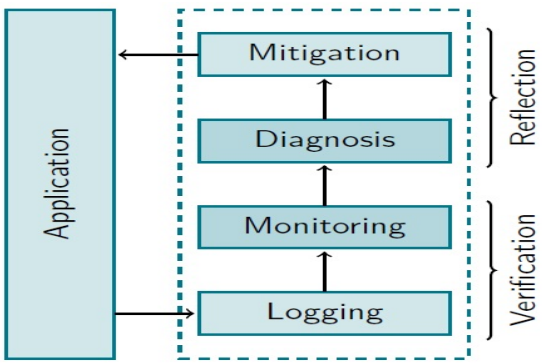


Fig. 2. An application and the layers of the runtime reflection framework.[11]

It consists of 4 layers, the role of the *logging layer* is to observe system events and to provide them in a suitable format for the monitoring layer. Typically, the logging layer is realized by adding custom code annotations (e.g., java annotations like @override used for instructing compiler that the annotated method is overriding the method.)[15] within the system to build. The logging layer allows to register so-called loggers, i.e., software components for observing the stream of system. While the goal of a logger is to provide information on the current run to a monitor, it may not assume on the properties to be monitored.
The *monitoring layer* consists of a number of monitors which observe the stream of system events provided by the logging layer. Its task is to detect the presence of faults in the system without actually affecting its behavior. Runtime reflection is assumed to be implemented using runtime verification technique. If a violation of a correctness property is detected in some part of the system, the generated monitors will respond with an alarm signal for subsequent diagnosis.
The *diagnosis layer* collects the verdicts of the distributed monitors and deduces an explanation for the current system state. For this purpose, the diagnosis layer may infer a minimal set of system components, which must be assumed faulty in order to explain the currently observed system state. The procedure is solely based upon the results of the monitors and general information on the system. Thus, the diagnostic layer is not directly communicating with the application.
In *mitigation layer* the results of the systems diagnosis are then used in order to reconfigure the system to mitigate the failure, if possible. However, depending on the diagnosis and the occurred failure, it may not always be possible to re-establish a determined system behavior. Hence, in some situations, e. g., occurrence of fatal errors, a recovery system may merely be able to store detailed diagnosis information for off-line treatment.

3 Background

In this overview we will take a look at current state-of-the-art methods of runtime verification their applicability to financial and medical application domains together with commonalities , advantages and challenges.

3.1 Financial Transaction System

The huge financial service providers such as Paypal, Revolut etc., facilitates easy transaction of money online across the globe. These companies are mandated to follow a set of regulations e.g., a user cannot transfer more than 2000 a week. Following these regulations becomes hard when track of billions of users has to be kept. By using software engineering processes the solution to this issue which happens at the payment gateway can be found. Today with the increase in software systems it is unthinkable to advocate and accept any software developed in an ad-hoc manner(for a particular situation), since it stands little chance of being adopted in the short term.

Runtime monitoring and verification have been advocated as very industry-friendly techniques,especially due to their scalability to large systems. Runtime verification technologies is adopted to improve the dependability of real-life systems. The formal verification techniques like testing, offline verification and static analysis tools have the luxury of being used prior to deployment, thus mitigating part of this problem. However, with online runtime verification, in which verification is performed during execution, this becomes an inevitable issue.

3.1.1 Basic Method

Ixaris Ltd., which planned to handle high volumes of financial transactions across different user applications and financial institutions has developed an industrial system to integrate runtime verification technology into the Open Payments Ecosystems(OPE). [2] Compliance to legislation and correctness are critical in this domain, and the risk of failure due to the runtime verification module had to be mitigated. For this purpose, Valour, a runtime verification tool was developed, it acts act as a back-end verification tool for financial transaction software.

3.1.2 Valour: Architecture and Design

Valour has been designed to work as a tool which processes streams of data received from (potentially) different sources. This is achieved in three stages.

- Trigger manager process triggers arriving from the systems being monitored, categorizing them and tagging them appropriately with relevant data to package them as *events*.
- Event manager which consumes the events produced by the trigger manager farming them to the appropriate monitoring unit and triggering new ones if required.
- Monitoring unit instances which include the logic as to how to react upon receiving a relevant event.

Since Valour allows for properties to be monitored for each instance of a particular type, it includes a *template instantiator* which creates a new monitoring unit whenever events pertaining to a new instance are received.

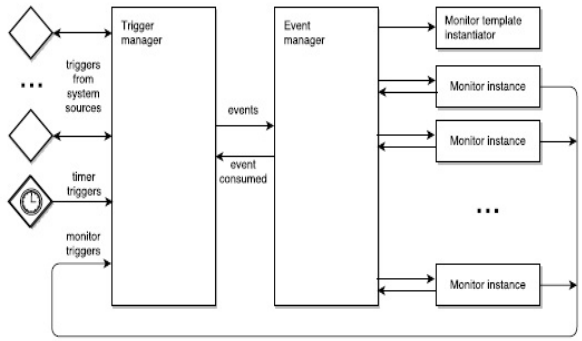


Fig. 3. Architecture of Valour[1].

The architecture of Valour in figure 3 is explained as below. **EVENTS:** In Valour, events are named and can be parameterised by a data tuple(Key, value pair) e.g. *closeAccount(destinationAccount)* corresponds to an event named *closeAccount*(key) carrying data parameter *destinationAccount*(value) corresponding to the account to which any remaining funds are being requested to be transferred. At the event level of abstraction it is not important whether this event comes from a method call in the main system or any other source.
In addition to parameters, events may be annotated by the categories they correspond to. For instance, *closeAccount* may be categorized by the class of bank accounts, or the category of users. This information, together with information of how to extract the category instance from the event, is used by Valour so that the event is passed on to the relevant monitor.
Triggers: Event Generators: Valour provides two constructs for defining system triggers coming from different sources:

- *controlflow trigger :*triggers corresponding to control points in the code of the system, and would be instrumented via aspect-oriented techniques[10].

- *external trigger :*allow valour users to define custom external triggers through an application programming interface(API) which in turn can be consumed by Valour and translated into events.

The separation of triggers from events helps keep the monitoring and verification specifications independent of the more technology-specific aspect of identifying points of interest during a systems execution.
Monitors: Event Consumers Monitors in Valour are written in a specific format using a guarded command language with rules which trigger when event *e* is produced, with boolean condition *c* holding, upon which action *a* is executed. We refer to a group of such rules as a *monitor.* There are two key features which enable the use of such rules for complex specifications:

*Monitor templates:*Many specifications are not global ones, but are to be instantiated for all instances of a particular category which might arise at runtime. In Valour this is supported by having monitor templates which quantify over a particular category of events, and are instantiated whenever a relevant event belonging to a new instance of that category is received. Internally, Valour ensures that events are only passed to the relevant monitors, thus reducing overheads in event consumption.

*Monitoring state:*Monitors may carry local state to keep track of information relevant to a particular rule block. This state can be accessed through the rules conditions and actions. For instance, a rule might be used to keep track of the running total of e-money transactions within an application, or to keep track of all users who transferred money to a particular geographical region over the past 24 hours.
Event Consumption Each event effectively instructs the Event Manager how to present monitoring results when the triggers generating the event are observed at runtime. With synchronous events, the manager holds off supplying the result of the event consumption until all computation is completed, while in the case of asynchronous events, an intermediate acknowledgement is immediately sent back, which can be used to check for completion status and monitor results. It is worth noting that the same trigger (e.g. a method call) can generate multiple events, thus allowing for it to be consumed synchronously in one property, and asynchronously in another. [8]

3.2 Medical Care Domain

In medical health care, decision making is not only a very critical task, the criticality also highly depends on whether the decisions are made on timely basis or not. For example, assume that a patient is showing signs of a disease, if these signs are detected early the patient could be treated early without any more damage. Currently Runtime verification is yet to be completely immersed in the medical domain, although it has been already tired in the medical doamain at a rudimentary stage already. In the following sections we discuss about, how runtime verification aids for making such critical decisions.
In Medical healthcare, a decision is to made using Decision Support System(DSS). A decision support system can be a monitor or a program that reflects the patient data on a screen that is then observed by the practitioner or the doctor. For example, a heart rate monitor is used to monitor the patients heart pulse. The heart pulse is then timely monitored by a doctor who will check if there are any signs of critical condition. In this case the heart rate monitor is the decision support system. It helps the health care providers to make better decisions based on multiple factors related to patients data. As medicine science develops, the real time data that needs to be sampled especially for some complex diseases quantifies to a large extend which challenges the decision support system. To reduce this complexity runtime verification is implemented on top of the decision support system. For introduction of the runtime verification consists of following components, one of them being A domain-specific language (DSL), it is a computer language specialized to a particular application domain and in this case it is the medical health care domain. In this paper, a domain-specific language named as (DRVT)[9] is used, DRTV specifies the vital real-time data that is sampled by medial devices, based on which event sequences are automatically created. Event sequences are a stream of events which are continuously monitored. If some property is violated in this sequence then the run-time verifier will produce real-time warnings which are passed to the decision support system, which are later observed by the practitioner.

3.2.1 Methodology

The basic workflow of how runtime verification works is represented in the below flowchart in figure 4. The electronic data of the patient is recorded and passed in the DRTV model which records the data format of these data and passes them to the run-time monitor, along with this the clinical guidelines(i.e. a set of rules that define if the patients data is outside the range of normal) is also passed to the DRTV model in the form of properties, the data formats and the data from the medical device sensor is passed to the Runtime Monitor. The Runtime monitor checks if there is any violation of the properties, if any violation is found an alert is sent to the decision support system. This decision support system is checked by the doctor.

This workflow can be illustrated with an example in figure 5 The patients blood pressure is observed, if it falls out of the expected range. The patients electronic data is collected from the medical device and passed to the DRTV model. The DRTV model generates an event

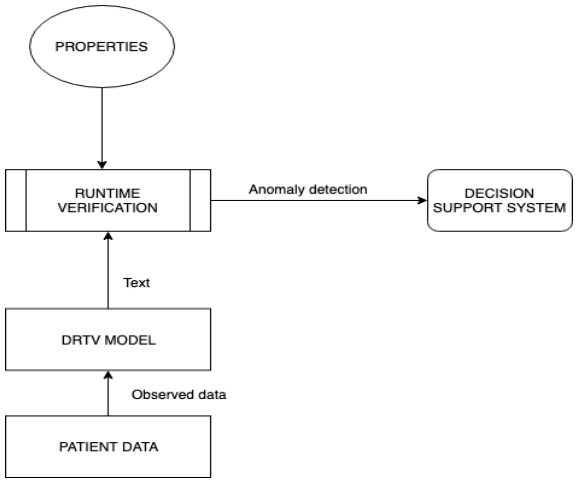


Fig. 4. Basic runtime verification flow

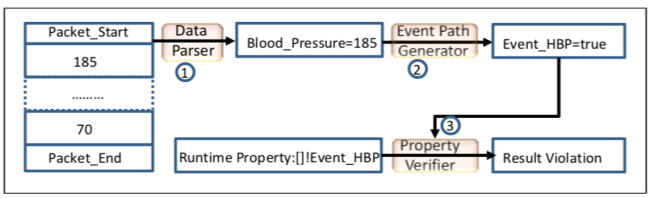


Fig. 5. A blood pressure example to illustrate the working of runtime verification[9]

which will compare the data and its corresponding format with the clinical guidelines, this comparison is sent to the property verifier, as per this example, the blood pressure observed is 185 which is above the threshold of 120 and possibly indicating high blood pressure, since the maximum threshold is 120 an alert is sent to the decisions support system that the result is violated. The practitioner observes these violations and takes the necessary actions.

Design The tool implementation describes a flow of how the system works with the DRTV model which is the basis for introducing run-time verification in the static or semi-automated decision making system for medical health care. The brief working of the same is introduced below and the representation can be referred in figure 6.

1. The main job of the DRTV model is to describe the data, events(i.e the change in the data) and time dependent properties in different scenarios, for example, create an event of 1 if the blood pressure of a person increases above a certain limit else return 0 describes a Boolean type data format and the blood pressure analysis is the scenario.

2. The scenarios are explained with the help of semantics and newly created syntax, the new syntax are created using ptLTL[14] and Arden. The ptLTL semantics is used for formulas specifications, it maps current state and the past states and compares if any property or condition is violated. While for Data specification part, some features and syntax from Arden(a domain specific programming language) were used, which is nothing but a clinical guideline modelling language.

3. Next, an interface for the DRTV model is constructed along with an engine which translates the model output in an XML format, the .XML format contains the data, the events and a property verifier. The data can be in any format, it can be Boolean, Integer etc depending on the scenarios. The scenarios here are nothing but the set of event that describe a scene. A scene is where a person might be observed for certain diseases and their activities

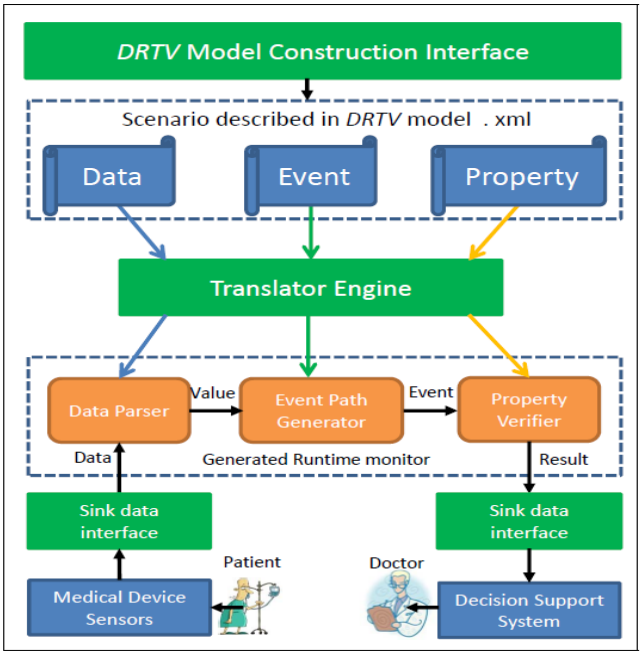


Fig. 6. The DRTV tool Implementation adapted from runtime verification to improve quality of medical care practise [9]

are monitored. Events demonstrates the actions which can or cannot show any significant changes, if significant changes are observed it means that violations are occurring. The properties are a set of rules that define the guidelines.

4. The translator which is a par of the engine (i.e Translator Engine) will translate the .XML files into executable java files which can be compiled on a computer.

5. This java file is passed to the verification model (i.e the Generated Runtime monitor). This consists of three parts, the data parser which abstracts the vital signs from the data received from the medical device or from the patients data. The Event path generator will use the values of the vital signs in the previous steps to evaluate a boolean formula to get corresponding events which can be useful to early detect any hazard. The property verifier will check if the automata transits into any violation state[9]. The Violations state is a state in which, a change in the expected value or range of value is detected and can further impact the decisions being made.

6. The last component of the model design is the communication interface sink which is used for data transfer from the medical device sensor to the decision support systems. Decision support system can then be observed by the practitioner.

4 Discussion

We discuss about a few benefits and challenges of using runtime verification in the application domains discussed above. We also try to trace out a few commonalities that were observed.

4.1 Commonalities

The paper discusses about how runtime verification was used in two different application domains. However, there were commonalities that were observed between these applications with respect to runtime verification. A few discussed below,

- Both the applications uses the common grounds of property verification technique in which an alert of some kind was sent when there was a violation observed on this real time data.

- In both the applications, decision making plays a very critical role, in financial aspect if any unwanted transaction is made and the end user is not informed, it can have some major consequences like, someone losing huge amount of money from their accounts, similarly in case of the medical scenario, if any vital signs of a major disease is missed out, it could lead to major consequences like, a disease not being diagnosed early. To sum it up, in both the scenarios it was observed that runtime verification made the system more reliable and better decisions could be made.

- Usage of monitors as a mode of observing fluctuation from the normal was common in both the applications.

- Even though there are semi automated techniques present, the workload of timely observations by humans was significantly reduced by runtime verification. This promoted a stress free environment in both the domains.

- Often, some information is available only at runtime or is conveniently checked at runtime. For example, when the library code with no source code is a part of the system, only a vague description of the behavior of the code might be available. In such cases, runtime verification is employed in the techniques.

4.2 Benefits

- Due to runtime verification the real time data are analyzed, and hence whenever there are an anomaly an alert was directly sent to the decision support system instead of the doctor constantly having to check for any abnormalities. Due to this the overhead on the practitioner was significantly reduced.

- As more complex a disease gets, more complex the guidelines for following it and checking for observing anomalies in a patients data gets, it was observed that for complex medical guidelines like infants respiratory distress syndrome, it was not easy for the semi-automated manual inspections visions [9] to monitor the infant and its conditions for the continuous 30 seconds steep time frame, but with the help of the runtime verification technique was constructed using certain syntaxes and semantics in which if there are 30 number of consecutive steep blood-gas incensement, it would indicate a violation of the desired property and a timely warning is responded immediately[9]. The hazards that the disease could cause was controlled and early signs of anomalies were noted way early.

- Now a days there are a lot of automated devices that can guide the physicians or for that matter any specialist to make a decision. These devices provide the physicians with information, but there is an extra set of dimensionality that is added to this information that can be misinterpreted by the physicians, due to a stressful environment, this can a lot of times be a cause for the information being neglected, and this can lead to ignorance of several warnings, but with the runtime verification, the physicians were much more relaxed as this was not a semi-automated technique.

- The tool has been integrated as part of a real-life financial transaction system in order to verify and monitor various elements, including legal compliance verification, risk-analysis for banks and service providers. Its use in the heavily tested environment of the OPE(Open Payments Ecosystems) has ensured that many potential issues with the system have been identified and addressed.

- In the case of systems where security is important or in the case of safety-critical systems, it is useful to monitor behavior or properties that have been statically proved or tested, mainly to have a double check that everything goes well: Here, runtime verification acts as a partner of theorem proving, model checking, and testing.

4.3 Challenges

There are quite a few challenges that are faced with using runtime verification in the current medical systems. A few are stated below.

- The ptLTL[14] formula specification in DRTV model may be easy for someone from computer science background to understand but it may be complicated for the medical staff as it includes syntax and sematics that are better understood by a programmer. It is better to provide some easier templates and improve the user-friendly of the language.

- The syntax and the semantics add a new learning dimension to the practitioners, and since most of the time they already have tight schedule it can be quite challenging to learn new things.

- Runtime verification is still in rudimentary stages of development in the discussed domains of financial and medical. There is yet a lot of development in these domains to come.

- Since this is an additional feature to support the current decision systems, there is a need to develop more existing decision support systems which aim at more specific applications and domains which is still a challenge, since people are still reluctant to adopt runtime verification.

- The major drawback of runtime verification is that the code will be running within the system of the programmer. if the code is provided publicly it will rise trust issues.

- The important challenge of run time verification is the involvement of humans. Starting from the initial meeting to testing there is a constant involvement of humans needed.

- Certain methods are used to extract the events from the system and communicate these events to the algorithm which verifies the events. Though this process seems to be easy it is quite challenging and each subsection influences each other..

5 Conclusion

We have discussed in detail about what runtime verification is. We evaluated current software verification technique of static verification technique, unfortunately due to increasing complexity of guidelines or the properties it becomes difficult to maintain a track of the consistency with these static verification technique, hence there is a coherent need to introduce runtime verification technique as a better alternative. We discussed how runtime verification manages the execution thread by using monitors and instruments. We discussed the framework of runtime verification in financial and medical health care in detail and how data is dealt on a realtime basis in both of these applications. How signals or alerts are sent when a violation is observed to indicate anomaly. We discussed about the commonalities that were observed in these application domains, one of the major ones being that since most of the data was handled by a software on realtime basis there was a stress free work environment. We introduced the benefits of using runtime verification for better decision making. Unfortunately due to various challenges involving runtime verification a complete realization of runtime verification in all the aspects of these domains still faces a lot of challenges which are discussed in the challenges section.

References

[1] Shaun Azzopardi, Christian Colombo, Jean-Paul Ebejer, Edward Mallia, and Gordon J Pace. Runtime verification using valour. In *RV-CuBES*, pages 10–18, 2017.

[2] Shaun Azzopardi, Christian Colombo, Gordon J Pace, and Brian Vella. Compliance checking in the open payments ecosystem. In *International Conference on Software Engineering and Formal Methods*, pages 337–343. Springer, 2016.

[3] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to runtime verification. In *Lectures on Runtime Verification*, pages 1–33. Springer, 2018.

[4] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime reflection: Dynamic model-based analyis of component-based distributed embedded systems. *Modellierung von Automotive Systems*, 2006.

[5] Ilan Beer, Shoham Ben-David, Cindy Eisner, and Avner Landver. Rulebase: An industry-oriented formal verification tool. In *33rd Design Automation Conference Proceedings, 1996*, pages 655–660. IEEE, 1996.

[6] Karl D Bilimoria, Banavar Sridhar, Shon R Grabbe, Gano B Chatterji, and Kapil S Sheth. Facet: Future atm concepts evaluation tool. *Air Traffic Control Quarterly*, 9(1):1–20, 2001.

[7] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[8] Christian Colombo and Gordon J. Pace. Industrial experiences with runtime verification of financial transaction systems: Lessons learnt and standing challenges. In *Lectures on Runtime Verification*, volume 10457 of *Lecture Notes in Computer Science*, pages 211–232. Springer, 2018.

[9] Yu Jiang, Han Liu, Hui Kong, Rui Wang, Mohammad Hosseini, Jiaguang Sun, and Lui Sha. Use runtime verification to improve the quality of medical care practice. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 112–121. ACM, 2016.

[10] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[11] Martin Leucker and Christian Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[12] John McCarthy. A basis for a mathematical theory of computation. In *Studies in Logic and the Foundations of Mathematics*, volume 26, pages 33–70. Elsevier, 1959.

[13] Charles Pecheur, Alessandro Cimatti, and Ro Cimatti. Formal verification of diagnosability via symbolic model checking. In *Workshop on Model Checking and Artificial Intelligence (MoChArt-2002), Lyon, France*, 2002.

[14] Scarlet Schwiderski and Gunter Saake. Monitoring temporal permissions using partially evaluated transition graphs. In *Modelling Database Dynamics*, pages 196–217. Springer, 1993.

[15] Daniel Tang, Ales Plsek, and Jan Vitek. Static checking of safety critical java annotations. In *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, pages 148–154. ACM, 2010.

An overview of prospect tactics and technologies in the microservice management landscape

Edser Apperloo[1] and Mark Timmerman[1]

[1] *University of Groningen - Computing Science - Distributed Computing*

Abstract—In the past years, there has been a trend towards using microservice oriented architectures rather than monolithic architectures in different industries. Microservices offer flexibility, reusability and are easier to scale than traditional software. The use of microservices introduces new issues regarding their management. Monitoring, health management and service discovery are still open subjects of research. Even though the field is progressing rapidly, constant changes in requirements together with large market shareholders prevent companies from taking advantage of the benefits of the microservice architectural style. In addition, it becomes increasingly difficult to assert that the microservices and their APIs implement the proposed contracts.

In this paper we provide an overview of the microservices architectural style and its existing technologies and heuristics regarding the two aforementioned challenges: (i) he management of large sets of microservices and (ii) asserting the implementation of proposed contracts by those services. We provide an overview of the current microservices landscape and identify the benefits and pitfalls, together with the key challenges, their implications and the set of technologies currently available to face these challenges.

We identified the following trends regarding the microservices landscape by doing a literature study: Current solutions make use of a form of external global management applications like Kubernetes, Amazon CloudWatch or Rightscale. As the set of services grows and the need for management and orchestration increases, the management application itself increasingly needs scalability. As microservices keep gaining influence the need for less vendor lock-in and distributed management will increase. In order to facilitate this we expect a future trend towards distributed management applications in which microservices manage themselves or few others in order to guarantee scalability and reliability. Problems that might be faced in this scenario include communication between these management services and a clear separation between the microservice logic and its management layer.

Index Terms—Distributed Management, Microservices, Microservice architecture, Overview

1 Introduction

Traditional monolithic architectures have most of the components of an application bundled in one place, or a few layers. In a microservice oriented architecture the different components are split up into different so-called microservices. These microservices are smaller independent components that are small applications in themselves. The microservices are responsible for a singular task, and are able to communicate with other microservices through application programming interfaces (APIs). The microservices have their own architecture, technology and platform, and they can be managed, deployed and scaled independently from the rest of the software architecture with their own release life cycle and development methodology [14]. Ideally, the microservices are completely distributed and independent, having no knowledge of the exact implementation or location of other microservices. Figure 1 shows two example architectures. One showing the microservice architecture (left) and the other showing the more traditional three-layered monolith architecture.

1.1 The adoption of cloud computing

Cloud computing has grown exponentially over the past years and continues to grow [7]. It has become the de facto standard for high demand distributed computing aimed at millions of users. In fact, 96% of the respondents to the State of the Cloud 2018 survey [7] were either using cloud or planning on doing so in the near future. The world has embraced the use of cloud computing and many researchers focus on the development of new architectures, patterns, consensus algorithms and security in order to ease the development on remote clouds. This can be deduced from the huge amount of developed software and tools in the past few years, of which many will be covered in upcoming sections.
The use of cloud computing has two main motivations: either to increase productivity or to decrease cost [10]. This paper claims the cloud offers fast self-service provisioning and task automation which can be continuously removed and deployed. Waiting time for developing dependent system elements, testing and production is decreased and the flexibility of the application as a whole is significantly increased. They claim the economical costs as a whole are reduced because of the upfront savings from the pay-per-use model and the fact that there is no need for the company to tend to the actual physical architecture, as this is part of the tasks of the cloud provider [12].

Bringing existing applications to the cloud remains a challenge [10] and much research is put into developing proper development cycles on the cloud platforms. This research has grown fast over the last years and it has been found that the microservice architectural pattern fits very well to cloud computing. Due to the distributed and highly decoupled nature of the microservices, it has become native to development on clouds. Even when microservices were a novel style, it was already found that the microservice pattern is often linked to cloud-based containers for deployment and dynamic management [6].

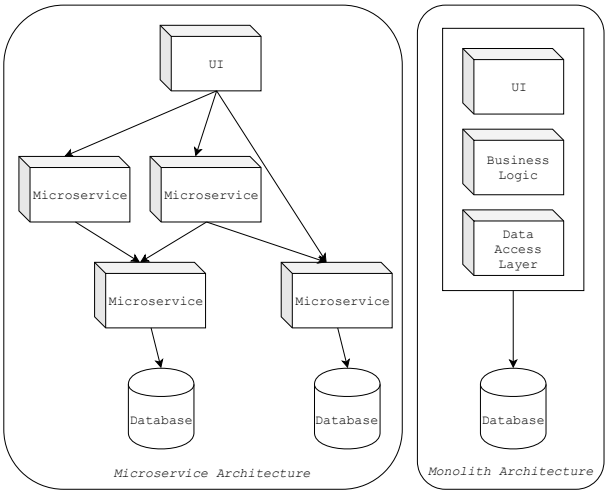


Fig. 1. Illustration of the microservice architecture versus the monolith architecture

1.2 Research contribution

This paper gives an overview of the microservice architectural style and the technology that is available to manage microservice applications. We review the development of the microservice architectural style over the years. This paper highlights innovations that were made which complement this architecture. Subsequently, it evaluates skepticism with which these innovations were met by seeing how those statements have aged. This paper tries to look past the hype [13], and critically reviews the capabilities and limitations of microservices. The microservice architectural style has matured significantly over the years, however, a definite bound of what microservices can and cannot do remains to be found.
As microservices have gained root, the academic world has invested considerable research into its capabilities. But where progress was made in, for example, security and consensus algorithms, many new challenges regarding management of microservices were uncovered in the process. We provide an overview of the benefits as well as limitations of microservices. It identifies key problems and proposed solutions. We will particularly focus on the challenge of managing a set of growing microservices and making sure that those services implement their proposed contracts. The general trend together with these key challenges are empirically reviewed and followed by predictions for its future.

The remainder of this paper consists of three sections. Section 2 describes the research methodology, section 3 describes our literary findings and their implications, and finally section 4 gives a conclusion to the results described in the sections before. This is followed by a short paragraph describing our expectations for the microservice trend in the near future.

2 Methodology

An overview of the current microservices landscape was created by performing a literature study on past-to-current research and examining the strategies used by software which is used to support microservice oriented architectures. An empirical overview of the benefits and challenges of using microservices was created. As the amount of microservices in an architectures grows, it becomes increasingly difficult to manage the microservice system. The amount of communication increases, and the system as a whole becomes more complex. With a larger system, it becomes increasingly important to ensure that the communication between the microservices is correct. Out of these observations, two key challenges in the microservices field were selected and more thoroughly examined. The first one being: What are the strategies and technologies available for managing an ever growing set of microservices? The second challenge involves the correctness of the communication between the different microservices: How can you ensure that the microservices implement their proposed contacts? The different tactics and heuristics related to these challenges were investigated, and mapped in this paper.

3 Results & Discussion

In the subsequent subsections, different aspects of microservices are thoroughly analyzed. Benefits and pitfalls are discussed, and possibilities in approaching those pitfalls are identified.

3.1 The benefits of microservices

The microservice architecture gives rise to many benefits for a system. Since microservices are independent from the rest of the application they are also independently deployable. If the services are designed correctly, then, different developer teams can work on different services without interfering. It creates a true divide and conquer environment that enforces loosely coupled, or even completely decoupled, components in a system as independently developed components cannot rely on other components of which the internals are unknown.
Internal changes can be made to microservices without influencing other microservices as long as the service itself keeps delivering the contracts proposed by its API. Contracts are the agreed upon design decisions dictating the functionalities that an API should implement. Making changes to one microservice should not influence any of its consumers as generally resources are not shared between microservices [13].
Microservices are small standalone applications that fulfill a particular task in a larger application. Often when demand of a system increases there is a specific part of the system that becomes the bottleneck, not the system as a whole. Whereas monolithic application design needs to scale the entire system because of a bottleneck in one of its components, the microservice architecture allows for upscaling only those microservices of which the demand has become the bottleneck. This saves money and is much easier to do as microservices are small applications by their nature.
Some of the microservices will be quite general in their task, which allows for reuse of parts of a system when new functionality is added or even for a service to be transferred to entirely new applications. Figure 2 shows a small e-commerce application written in terms of microservices. Here one can see the decoupled nature of microservices. The device specific frontend services depend and reuse the same services further towards the backend. Each of those has their own database again, such that in case of extreme failure not all data is lost.
Yet another benefit to microservices is that they eliminate any long-

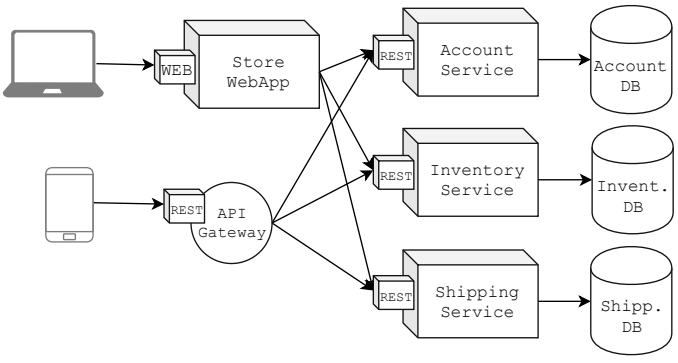


Fig. 2. An e-commerce application in terms of microservices

term commitment to a technology stack. Every time a new service is developed, one can choose a new stack for that particular service. Software engineering is a field in which new frameworks, programming languages and other ideas are developed every day. A system that is aimed at lasting throughout the years needs this flexibility in order to keep up with the pace and developments of its field. This means

that developers are never bound to use a particular stack throughout the life cycle of the system. Since microservices are smaller, maintenance becomes straightforward and legacy code is more easily ported to a new stack.
The aforementioned benefits all allow for easy development, deployment and scaling of each individual microservice. Microservices also show another property that makes them scalable and lasting: recursion. Building a system comprising of smaller systems reeks of recursion.It allows for applying the same mechanism again when the requirements of systems become even more complex and are asked to perform even more different tasks at once. In other words, when the requirements change, instead of modifying the system, one can build a new system that adds the newly requested features. The two systems now make up an even larger system.

This notion of recursion in microservices even has some roots in for example biology. Combined with self management and with placement across failure domains, some proposed architectures (using microservices) enable distributed hierarchical self management, akin to an organism (i.e. the composed system) that is able to recreate its cells to maintain its morphology while each cell (i.e. a microservice) is a self-managing element [10].

3.2 Vendor lock-in

Microservices have limitations, however not all of these are inherent to the design of this architectural style. One of these is vendor lock-in, it is prevalent in the microservices landscape since there are only a few large providers. Vendor lock-in is the notion that companies are not able to transition easily to other providers. As companies transition their applications from monolithic architectures to microservice architectures it is almost imperative to use the large providers their services in making the transition. Choosing one of these providers usually means sticking to them, as the cost of migrating from one provider to another has become difficult and expensive. Before long, it becomes too expensive to switch for many companies, resulting in vendor lock-in at their current providers. Businesses are wary of being tied to particular cloud computing vendors due to lack of competing, compatible product [5]. This is a significant step in the wrong direction as the microservice architecture relies on its flexibility and technology independence. That is, really picking the right tool for the job. Due to vendor lock-in this often prevented. Even the European Network and Information Security Agency (ENISA) and European Commission (EC) have recognized the vendor lock-in problem as one of the greatest obstacles to enterprise cloud adoption [3]. Since there are only a few technologies available for managing microservices, and only a few companies providing cloud services, these players have gained an enormous amount of power and control in the cloud industry.

3.3 Challenges and disadvantages of microservices

On the one hand the use of microservices reduces complexity and increases agility and scalability of those microservices. However, the usage of microservices also introduces challenges.

3.3.1 Complexity

However, the added complexity of coordinating and creating a system of communicating microservices introduces a lot of complexity as well. More configuration to coordinate microservices is necessary and this can overwhelm the relatively small gain received through the use of microservices in the first place [13]. Many of the problems from distributed computing are inherited such as data synchronization and reaching consensus. These problems are aggravated by the very distributed and dynamic nature of microservice applications. The utopia that microservice enthusiasts picture is in many practical cases just that, a utopia.

3.3.2 Coupling

While it is possible to design a microservice system very well, it is always necessary to make compromises. The slightest unforeseen coupling between different microservices can give rise to a waterfall of elements that need to be taken into account when a single, supposedly uncoupled, service is updated or somehow changed. Developers then have to run or connect to other microservices when they are working on a service dependent on other services.
A recent study has suggested however, that once the application has made the complete switch to a well designed system using microservices, that complexity grows significantly slower than for systems that have not made this switch [8].

3.3.3 Transitioning

The rapid demands of the digital world make it hard for companies to make the switch towards microservices from legacy code, as rewriting codebases from monolithic applications to microservice applications can be difficult [1, 13]. Redesigning and recreating entire parts of a system takes time which could be spend on implementing new requirements instead of porting old ones. In the current digital world, where demand grows quickly and new requirements are constantly added, this makes sure that a business looking to make an architectural pivot like this is not just standing still, it is falling behind. While the company focuses on pivoting, competitors can rush to market fulfilling those new requirements and the pivoting company is left behind. For such cases either intermediary or other solutions are needed.
Many cloud providers or third party services help to overcome this issue by offering as much support as possible. One could think of monitoring tools, health care management and automated scaling. Large market technologies include Amazon CloudWatch[1], Auto Scaling[2], Kubernetes[3] and Rightscale[4]. Most of the possible solutions are steps that lead towards vendor lock-in [10]. Where microservices excel in the the fact that you are not bound to any technology stack, the vendor lock-in introduced by so many providers prevents this style from achieving its maximum potential.

3.3.4 Management and Orchestration

As mentioned before, decoupling of legacy code and creation of microservices to reduce complexity of the entire application introduces complexity itself: management of the distributed services as well as their communication and orchestration. Even more problems than those may arise as components communicate via message passing and through APIs, which in turn may change over time. The fast paced digital world does not allow for companies to spend proper time on this communication and management issue as they will be outperformed by competitors since no real advancements are made from the user perspective. The management and communication of microservices is an on going research topic and one of the key challenges for this architectural style. Many strategies and tools have been developed to deal with the management and orchestration issues as well, however it is not always clear what the best tools are for the job. Local versus remote testing is hard and annoying and a lack of proper tool support creates a situation in which it becomes hard to navigate towards a proper set of tools. [6].

3.4 Tactics to manage large sets of microservices

Managing large sets of microservices is a difficult task because of the amount of microservices that need to coordinate their communication. To help manage these microservices, different technologies and tactics have been developed. An overview of the different technologies can be found in table 1. The upcoming sections elaborate on the different tactics developed for managing microservices.

Research has shown that the larger challenges of microservices concern one of these tasks and a lot of effort has been put into developing new architectures and patterns to help with these challenges. A study by Toffetti et al. [10] describes the following issue with the plain microservice architectural style:

[1] https://aws.amazon.com/cloudwatch
[2] https://aws.amazon.com/autoscaling/
[3] https://kubernetes.io/
[4] https://www.rightscale.com/

> *"In the current practice, management functionalities are provided as infrastructural or third party services. In both cases they are external to the application deployment. We claim that this approach has intrinsic limits, namely that separating management functionalities from the application prevents them from naturally scaling with the application and requires additional management code and human intervention. Moreover, using infrastructure provider services for management functionalities results in vendor lock-in effectively preventing cloud applications to adapt and run on the most effective cloud for the job."*

The work of Toffetti et al. is now a few years old and many developments have been made in order to prevent the mentioned issues with microservices. Results show however that vendor lock-in is still a relevant issue in the world of cloud computing and microservices [9].

3.4.1 Service discovery

In monolithic architectures, most of the components are in a few layers and each consumer knows where to find its producers. Microservices need to communicate with other services, so it is necessary for microservices to know where to find the other microservices. Before service discovery technologies were available, the microservices themselves were responsible for keeping track of the addresses and locations of other services. This means that the source code for discovering other services plus the handling of possible failures was embedded in the source code and logic of the microservices themselves. In order to be fully independent and decoupled it is necessary for microservices to abstract this away. Interfaces should be discoverable: consumers must be able to look up interfaces of the producer without having explicit knowledge of the underlying technology, implementation or location [13]. For this, the concept of service discovery is introduced [2]. Microservices are able to register at a discovery service, and other services are then able to access the microservice through this discovery service without an explicit reference to the microservice itself. Service discovery technologies are often implemented as distributed key-value stores. Examples of technologies which provide service discovery are: etcd[5], Zookeeper[6]. Netflix Eureka[7], Synapse [11] and Consul[8]. Typical usage of services like etcd is to automatically update configurations whenever there is a relevant change in the system (e.g. when a new back-end server is spun up the load balancer needs to distribute its load also to this newly registered server) [10]. Figure 3 (a) depicts how communication is done if there is no discovery service available. All of the communication logic is embedded in the business logic. When a discovery service is added (3 (b)) the services are able to discover other microservices dynamically.

3.4.2 Sidecars

Sidecars are introduced as service intermediaries. In sidecars, all service discovery and communication are encapsulated so that the business logic is isolated. Most of the fault-tolerance is delegated to a dedicated unit attached to the microservice. This allows for further separation of business logic and communication logic between services. Another advantage of using sidecars is that existing tried and tested fault-tolerant communication packages can be combined with any microservice as sidecars. Some examples of sidecar technologies which are used in the industry are Envoy[9] or Netflix Prana[10]. Figure 3 (c) shows the communication paths between microservices when sidecars are added.

A standalone sidecar does not fulfill much purpose as a microservice without business logic is not useful, nor is a microservice that does not communicate with others to complete a common goal. However, if multiple sidecars from microservices are combined, a mesh of sidecars with attached microservices is created. Those so-called service meshes are able to keep track of traffic flow and route traffic between microservices. Service meshes also fulfill the purpose of circuit breaking. Circuit breaking is limiting the impact on the whole network if one of the microservices is malfunctioning. Besides circuit breaking, service meshes are able to handle failure of microservices gracefully when a microservice is unreachable, and thus ensuring that errors do not propagate through the communication paths. Some technologies which implement service meshes are Linkerd[11], Istio[12] and Conduit[13].

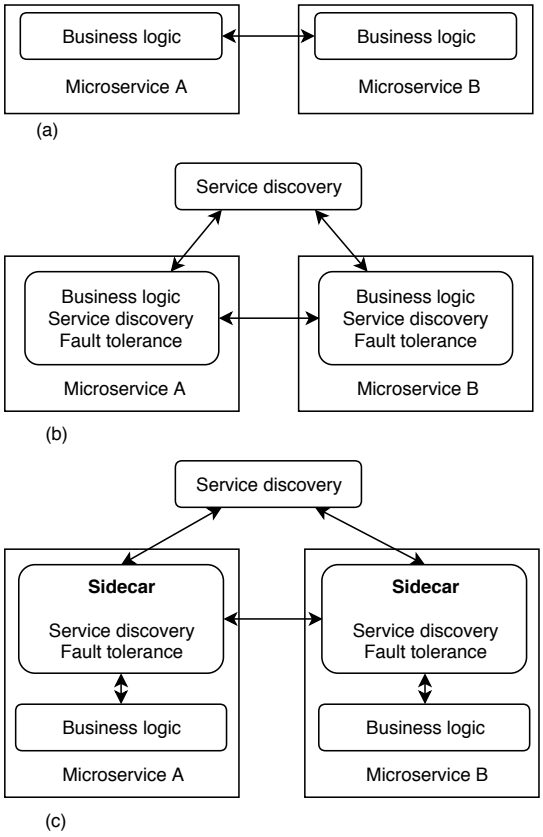


Fig. 3. Different types of service discovery. Figure a denotes simple communication between microservices A and B. In figure b, a discovery service is introduced which services can use to discover other services. In figure c, sidecars are added to the microservices which are responsible for the communication between the microservices.

3.4.3 Orchestration

As the amount of microservices in a system grows, it becomes increasingly difficult to manage the available resources manually. Orchestration technologies have been developed to automate resource allocation and management tasks [4]. The orchestrator makes sure that the resources and configurations for each service are available when necessary. A single centralized unit is added which is called the orchestrator. Every time an instance of a service is created it registers itself at the orchestrator. Now when a consumer needs to find a producer it uses the orchestrator to find a suitable consumer to complete the transaction. Examples where global service orchestration like this

[5]https://coreos.com/etcd/
[6]https://zookeeper.apache.org/
[7]https://github.com/Netflix/eureka
[8]https://www.consul.io/
[9]https://www.envoyproxy.io/
[10]https://github.com/netflix/Prana
[11]https://linkerd.io/
[12]https://istio.io/
[13]https://conduit.io/

is used are platforms such as Kubernetes[14] or Docker Swarm[15]. Table 1 shows different technologies which are used for orchestration.
This global form of orchestration is also being researched as studies show that having such an abstraction from the services themselves introduces complexity in scaling and managing of the orchestration software itself [10]. The proposed solution there is to make the orchestration software part of the microservice itself, such that they become self managing and scaling of the system becomes natural with the scaling of microservices as they scale their own management as well.

3.4.4 Monitoring

It is important to keep track of key metrics in a system, especially in microservice oriented architectures where a lot of communication is happening. Distributed request tracing allow you to profile the system and discover bottlenecks. Proper monitoring tools allow developers to improve and debug the system effectively. Collected metrics can be used to create dashboards, which reflect the health of different microservices. This allows system administrators to make informed decisions about how to improve the system. Live health management is important for distributed systems as proper monitoring can increase availability. In this age, any availability below 99% is considered poor, so there is high demand for services providing such health care management. Some technologies which enable this type of monitoring are Prometheus[16], InfluxDB[17], Graphite[18], Sensu[19] or cAdvisor[20]. The monitoring technologies are often combined with service meshes in order to track the traffic between the different microservices.

3.5 Contracts between microservices

The contract of an API of a microservice provides information about the functionalities it employs. It is the gateway to a service such that consumers can use the API to fire encapsulated code. Microservices can change the code under the hood freely without influencing its consumers as long as the promises an API makes are delivered upon. When changes need to be made to the APIs themselves it becomes critical to assert that contracts are not broken by the adjustments. Different heuristics exist to improve API design. API Blueprint[21], Swagger[22] and Apiary[23] are examples of tools that help reducing design errors and enforce API contracts to be held up. This helps in reducing development time and facilitating future changes without breaking any of the older technologies. However there are more ways to keep APIs from breaking promises they describe. Tools such as Pact[24] and Spring Cloud Contract[25] are contract testing tools such that APIs can be properly tested without the need for integration tests.

[14]https://kubernetes.io/
[15]https://docs.docker.com/engine/swarm/
[16]https://prometheus.io
[17]https://www.influxdata.com/
[18]https://graphiteapp.org/
[19]https://sensu.io/
[20]https://github.com/google/cadvisor
[21]https://apiblueprint.org/
[22]https://swagger.io/
[23]https://apiary.io/
[24]https://docs.pact.io/
[25]https://spring.io/projects/spring-cloud-contract

Table 1. Microservice tooling landscape

Technology category	Technology name
Service discovery	etcd, Zookeeper, Netflix Eureka, Synapse, Consul
Sidecar	Netflix Prana, Envoy
Service mesh	Istio, Linkerd, Conduit
Orchestration	Kubernetes, Docker Swarm, Amazon ECS(Elastic Container Service), Mesos, Rightscale
Monitoring	Prometheus, InfluxDB, Graphite, Sensu, Amazon Cloudwatch, cAdvisor, Rightscale
API development	API Blueprint, Swagger, Apiary
Contract testing	Pact, Spring Cloud Contract
Cloud providers	AWS, Google Cloud, Microsoft Azure, IBM Cloud Services, Salesforce

4 Conclusion

Microservices are small applications that, when composed, form larger systems that complete business requirements. Microservices are independent from other microservices and allow different technology stacks, platforms and deployments cycles to be used. They communicate through interfaces called APIs that implement the contracts defined by the microservice itself [14]. Unfortunately, the true potential of microservices often cannot be reached because of a number of issues. Firstly, it is not always feasible for companies looking to make the switch from monolith code bases. Secondly, pivoting to a new architecture is very time consuming and costs a lot of money. Most companies cannot afford this as they will be outperformed by competitors too soon. A plethora of available tools are available to help, but in turn worsen the vendor lock-in which is one of the other major issues for microservices. They can run on separate deployment cycles and technology stacks, however due to the large share of a few companies in the business, many options tailor towards using their services. This limits the flexible nature of the microservices and does not always allow for picking the best tool for the job. Microservices only work when executed properly, because minor design flaws can give rise to major complications when making changes to the system. However, when done properly, microservices offer a flexible, fast, and easily scalable solution that, partially due to its recursive nature, could potentially grow to systems of arbitrary size as the digital world keeps demanding more and more of its systems. The current research community is investing a lot in a few key challenges. The current limits faced by systems using microservices involve orchestration, health management and communication [10].

As the knowledge about microservice oriented architecture progresses, more tools are being developed to manage applications consisting of microservices. Microservices are reliant on communication between them, and thus it is important that the microservices are able to know the locations of the other services. Service discovery tools like etcd are used by microservices to dynamically discover the location of other services. Another important trend is that the business logic of microservices is being isolated from the communication logic. Sidecars are a way to use tried and tested communication layers in conjunction with the business logic of a microservice. When each microservice has an attached sidecar responsible for communication between microservices, service meshes can be created. These meshes are able to route and keep track of traffic between the services, and are able to function as circuit breakers if failures occur somewhere in the service mesh. Monitoring is an important factor in management of microservices. Key metrics such as performance, bugs and health can be logged. In addition to this, distributed tracing can be used to debug and improve the microservice architecture. When sidecars responsible for communication between microservices are added to each microservices, monitoring of communication is trivial. All of the collected metrics can be

used to create overviews and where administrators can easily learn about the health and bottlenecks in the system.

4.1 Expectations for the future

As the use of cloud solutions will only grow in the future we expect only more increase in the use of microservices or minor variations. Current cloud applications make use of external global management applications like Kubernetes, CloudWatch or Rightscale. As the set of services grows and the need for management and orchestration increases, the management application itself increasingly needs scalability. As microservices keep gaining influence the need for less vendor lock-in and distributed management will increase. As the maximum potential of microservices is yet to be reached, we expect the future developers to realize that this extreme form of vendor lock-in cannot continue infinitely. We think it therefore crucial to invest in developments steering away from the vendor lock-in currently seen in the cloud environment. In order to deal with problems that will inherently arise from this, we expect a future trend towards distributed management applications in which microservices manage themselves or few others in order to guarantee scalability and reliability. Problems that might be faced in this scenario include communication between these management services and a clear separation between the microservice logic and its management layer. Architectures as proposed in [10], show promising first results, but show challenges that come with the way the current IaaS providers have structured their services. Other research is done into different variations of the architectural pattern, but most of them lie on the foundation that microservices have created in the world of distributed cloud-enabled software applications.

REFERENCES

[1] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner. From monolith to microservices: A classification of refactoring approaches. *CoRR*, abs/1807.10059, 2018.

[2] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8. Boston, MA, USA, 2010.

[3] N. Loutas, E. Kamateri, F. Bosi, and K. Tarabanis. Cloud computing interoperability: the state of play. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science*, pages 752–757. IEEE, 2011.

[4] M. Mazzara and S. Govoni. A case study of web services orchestration. In *International Conference on Coordination Languages and Models*, pages 1–16. Springer, 2005.

[5] J. Opara-Martins. Taxonomy of cloud lock-in challenges. In *Mobile Computing-Technology and Applications*. IntechOpen, 2018.

[6] C. Pahl and P. Jamshidi. Microservices: A systematic mapping study. In *CLOSER (1)*, pages 137–146, 2016.

[7] Rightscale. State of the cloud report, 2018 Q2.

[8] N. Saarimäki, F. Lomio, V. Lenarduzzi, and D. Taibi. Does migrate a monolithic system to microservices decreases the technical debt? *arXiv preprint arXiv:1902.06282*, 2019.

[9] N. K. Sehgal and P. C. P. Bhatt. Future trends in cloud computing. In *Cloud Computing*, pages 171–183. Springer, 2018.

[10] G. Toffetti, S. Brunner, M. Blöchlinger, F. Dudouet, and A. Edmonds. An architecture for self-managing microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud*, pages 19–24. ACM, 2015.

[11] N. Viennot, M. Lécuyer, J. Bell, R. Geambasu, and J. Nieh. Synapse: a microservices architecture for heterogeneous-database web applications. In *Proceedings of the Tenth European Conference on Computer Systems*, page 21. ACM, 2015.

[12] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, et al. Cost comparison of running web applications in the cloud using monolithic, microservice, and aws lambda architectures. *Service Oriented Computing and Applications*, 11(2):233–247, 2017.

[13] Z. Xiao, I. Wijegunaratne, and X. Qiang. Reflections on soa and microservices. In *2016 4th International Conference on Enterprise Systems (ES)*, pages 60–67. IEEE, 2016.

[14] Y. Yu, H. Silveira, and M. Sundaram. A microservice based reference architecture model in the context of enterprise architecture. In *2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, pages 1856–1860. IEEE, 2016.

Dynamic Updates In Distributed Data Pipelines

S.J. Mallon, N. Meima

Abstract— This paper investigates the current ecosystem for performing dynamic software updating in distributed data processing pipelines. A distributed data processing pipeline is a set of combined elements which perform an operation on the data in parallel. The amount of data that is collected grows each year, and the need for proper distributed data processing tools continues. There are some major concerns when performing dynamic software updating to distributed data processing pipelines. Changing either the data source or the actual elements of the pipeline should not interrupt a running pipeline, since that would lead to potentially costly downtime. We evaluate general implementations of dynamic software updates in distributed systems, and also specific implementations for data processing frameworks like Apache Spark. Finally, the paper proposes concrete steps to undertake in order to mitigate eventual fallout from performing updates.

Index Terms—Distributed Data Processing Systems, Dynamic Software Updating, Distributed Systems

1 Introduction

Due to an every increasing amount of data being collected, the need for more advanced data processing tools increases. Through the growth of the Internet-of-Things sector (IoT), rampant user-data collection, and increasingly cheaper data storage hardware, the amount of data collected is growing fast. However, this ever increasing amount of data leads to complications when trying to perform the processing of said data on a single machine: perhaps a single machine does not have enough disk space to store the entire dataset on, or maybe the workload is too computationally intensive. Additionally, faster data processing might also lead to advantages, as it allows businesses to have up-to-date analytics, which allows to respond to changes in a faster fashion, giving them an edge over competitors. The processing of any dataset in the modern world thus necessitates means for faster processing. A solution which is often employed when there is a need for faster processing is the use of parallelization, in specific, the use of distributed systems. The workload is split over multiple machines, working in parallel, where each machine process a part of the dataset. After all machines have finished processing their respective parts of the dataset, the results are joined together. This distributed way of data processing introduces a new set of problems for data scientists and system engineers to solve: how can one design a system which can process large amounts data concurrently in a distributed way?

Here we provide an overview of state-of-the-art distributed data processing techniques and their dynamic software updating functionalities. It is found that current popular distributed data processing frameworks have limited support for dynamic software updating [19, 4, 8, 5]. In Dynamic Software Product Lines (DSPs) an update only entails switching between pre-defined elements in the pipeline, which is useful in some cases but does not allow for adding additional elements to the pipeline after deployment. Also, tools *spark-dynamic* use a wrapper approach, creating a layer around the distributed data processing framework for does allow adding additional elements to the pipeline after deployment. Although the aforementioned solutions are not all-encompassing and need further research, they do clearly show the usefulness of dynamic software updating functionalities. If a pipeline can be updated whilst running, this reduces downtime which saves resources.

1.1 Data Processing Pipeline

A data processing pipeline is a term to indicate a set of data processing 'elements', where the output of one element is connected to the input of another. The term pipeline arises from the sequential nature of the data transformations in directed acyclic graphs (DAGs). A data processing element in a pipeline could, for example, refer to a function that transforms the data in a meaningful way, or a function which stores the transformed data into a database. Imagine an example, where a user performs research on climate change and is measuring temperature as a function of time. The raw data is recorded in Fahrenheit. However, before doing any kind of data analysis, it is necessary to transform the data from Fahrenheit to Celsius. In the beginning of the pipeline, we could have an element that transforms the temperature data from Fahrenheit to Celsius, and afterwards, the data can be processed by the subsequent elements of the pipeline. A pipeline is a sequential set of transformations, which means that distributed computing cannot help performing these elements in parallel, as we need a fixed order of operations. However, it is very much possible to process the data in parallel. In general, we could refer to any data processing program as a pipeline.

The data processing pipelines in frameworks like Apache Spark have to be designed up-front after which they are submitted as processing jobs to the framework's controller [1]. This controller has the task of distributing the submitted job over workers. These workers each take care of processing part of the submitted processing job, realizing distributed data processing. After the submission of a processing job, the workers will be in a running state, and the process and data are immutable. This immutability is a limitation that comes with the use of such distributed data processing frameworks. However, there are many real-world scenarios in which the ability of dynamically changing elements of running pipelines is required. For example, a developer might find an error in the code of a stage in the pipeline which has not been reached yet. Instead of discarding the intermediary results of the earlier stages in the pipeline by submitting a new processing job in which the error no longer exists, an improvement might be to allow a dynamic software update to the stage of the pipeline affected by the error. Also, it might be required that new functionalities are added to the pipeline without interrupting the processing of incoming data, i.e. the pipeline cannot be shutdown temporarily. For example, the documentation of the distributed data processing framework Apache Spark, only proposes two mechanisms for updating application code of running pipelines [2]. The first option is to start a new processing job with the updated application code and run it in parallel to the existing processing job. Once the new processing job is ready to receive data, the previous processing job can be shut down. The data is then redirected to the new processing job. A downside to this approach is that the resources required for running a second, possibly complex, resource intensive and expensive, processing pipeline in parallel with the already running pipeline, must be available. The second option is to gracefully shut down the already running processing job. A graceful shutdown entails that the data that has been received has been processed completely before shutdown. After shutdown, the new

- *S.J. Mallon, E-mail: s.j.mallon@student.rug.nl.*
- *N.Meima, E-mail: n.meima@student.rug.nl.*

[1] https://spark.apache.org/docs/latest/submitting-applications.html

processing job can be started, picking up from the same point as where the other application left off. This approach leads to inevitable downtime, which might be very costly and not at all an option for some use cases.

2 Background

In this overview we will take a look at current state-of-the-art methods for performing dynamic software updates in distributed systems and their applicability to currently popular distributed data processing frameworks, together with advantages and points of caution.

2.1 Dynamic Software Updating

Dynamic software updating (DSU) can be compared to changing gears in a running engine. When applying a dynamic software update, the processes that are being updated do not have to be stopped, allowing them to preserve their progress, state and connections with other processes. Using DSU thus prevents the potentially costly stopping of the to be updated processes.

2.1.1 Dynamic Update Types

Endler and Wei proposed two types of dynamic updates which have been identified as necessary for most applications in need of DSU functionality: ad-hoc and programmed reconfiguration updates [7]. The ad-hoc dynamic update type entails designing updates while the process is already running. The designed updates are then configured using an interactive reconfiguration manager. The programmed reconfiguration dynamic update entails pre-programming alternative modus operandi into the application, which can be switched between during runtime. The most interesting dynamic update type is the ad-hoc type, which allows the most freedom and is the most viable strategy when compared to the programmed configuration type. Often, it is not possible to think of all software requirements in advance or requirements simply change due to external factors.

2.1.2 Dynamic Update Scopes

Dynamic software updates can also have different scopes. Most systems supporting DSU functionality do so by providing functions as a unit of code, meaning that only functions can be updated at runtime. However, there also exist DSU systems which support updates of the complete code running on the system. Which kind of updates the DSU system supports depends heavily on the environment of the system in combination with the software implementation.

When functions are the unit of code and the software implementation makes use of programming languages which compiles to machine code (e.g. C and C++), DSU functionality can often be realized by using a specialized compiler [14, 10]. The specialized compiler then adds a point of indirection in the code. In C or C++ such a point of indirection can be regarded to as a function pointer. A function pointer points to an address in memory containing the start of of a piece of executable code, like a function. The DSU functionality is then realized by updating the point of indirection is to the latest version of the executable code when an update arrives. Such solutions are often platform dependent, since they require specialized compilers which compile to machine code. This increases the difficulty and inflexibility of such a solution.

The larger the scope of the dynamic software updates become, the more sophisticated the environment needs to be. For example, if the unit of code grows larger than just functions and the software implementation is done in a programming language targeting virtual machines, the sophisticated VM can providing existing infrastructure to realize DSU functionality. The Java programming language is such a language targeting a virtual machine. A Java Virtual Machine (JVM) supporting DSU functionality is the HotSpot JVM [2]. Compared to the previous solution, the targeting of a VM provides a more platform agnostic solution, since the VM runs as a separate layer on the infrastructure. This makes the solution less complex and more flexible, since we have full control over the memory of the application at any time. However, the developer is still bound to certain programming languages and virtual machines. There exist also systems which are more language agnostic. For example, Katana realizes this agnosticism by operating on the level of ELF binaries [17]. On the contrary, there also exist complete environments supporting DSU functionality. The earliest example is the *dynamic modification systems* (DYMOS) [15]. the DYMOS DSU system consists of a complete environment for writing programs in a derivative of the Modula (which on its own is a derivative of the more well known Pascal language) language. The environment comes with all necessary tools, like: a command interpreter, a compiler and a runtime environment. The DYMOS environment also makes promises about and enforces type-safety of updates by inspecting the current software that is running.

[2] https://www.oracle.com/technetwork/java/javase/tech/vmoptions-jsp-140102.html

2.2 Dynamic Software Updating in Distributed Systems

Updating a single instance of a running application on a single machine is a well-researched and documented phenomenon [14, 10, 15, 17]. However, with the rise of distributed computing and a large increase in the size of the data to be processed, these solutions do no longer fit the modern requirements.

In turn, because of the growing need and adaptation of distributed computing, it is an active field of research [18]. In grid computing, where we have multiple machines performing an identical task, a solution to the problem was proposed: send updates simultaneously to all machines, along with a timestamp of when to start the specific changes [6].

However, in Spark's case, multiple machines do not necessarily perform an identical task: they perform heterogeneous tasks. However, this can still be resolved by keeping track of the state of all the machines in the grid, and only perform the update for the machines that are performing the relevant task to be updated.

When distributed data processing workloads are migrated from the grid to the cloud, the ability to directly interface with the machine the application is running on is severely limited. Only a limited amount of direct modification is feasible, like changing the database schema on the fly [13]. The benefit of these cloud providers is that they allow dynamic and automatic provisioning, which allows users to focus on solving the problem at hand instead of dealing with the secondary problem of managing infrastructure.

With systems like Apache Spark, a specialized distributed processing platform, update methods are either very limited or not applicable at all. The simplest way to update the data processing pipeline is to first stop the pipeline and then restart the pipeline again with the newest version of the application. If availability is a premium, this would mean that whenever an update is necessary, a second data processing pipeline needs to be active so we can switch with limited downtime. However, if the data processing pipeline is very resource intensive, this solution can be very expensive. In a worst case scenario, this is not possible at all due to the data processing pipeline requiring more resources than allocated, for example in a private cloud. If availability is not a premium, it should be considered that simply restarting a pipeline might be the best solution in such a use-case. Dynamic updates of the pipeline are complex, and a pipeline with a very short run-time might just as well be restarted, resulting in a simpler updating process. Only processes which are running permanently or those which have a long execution time are justified to be updated dynamically.

It is also possible to create DSU functionalities by establishing variation points in the program [9]. At runtime, the user can choose between a set of pre-defined implementations of certain parts of the pipeline (perhaps through a configuration file, or a GUI). However, this does not solve the problem of wanting to introduce a new type of algorithm into the dedicated pipeline. To be more precise: this would only work for situations which the user anticipates to happen and has created pre-defined elements for.

2.2.1 Cloud Web Solutions

Cloud Web orchestration solutions like Kubernetes also provide infrastructure to limit downtime and tools that lead to smoother rollouts of new software versions. Kubernetes allows for an (almost) seamless transition between versions: a container with the new application can be started, which then replaces an older version, after which the older version is terminated. This allows for minimal downtime for any users of the software.
The use of containerized applications is also helpful in rolling back to a previous version if the current version of the application does not function properly: we re-specify which version of the software is used to create an active instance. Such solutions are relatively easy to implement when the application controlled by the orchestration framework is ephemeral. Data processing frameworks are inherently stateful, making it harder to use similar orchestration techniques as a means of dynamically updating distributed data processing pipelines.
Additionally, one can choose to perform a red/green update, where a percentage of your application instances run on a newer version, and a set of application instances on an older version. Such functionality could be useful in distributed data processing pipelines. For example, when you want to update an element of the pipeline. However, this would require complete control over the element which is being replaced and segmentation of all elements in the pipeline: all elements should be independent of other elements. Such an approach is shown by OpenWhisk Composer, where serverless functions are arranged in compositions [3]. The OpenWhisk Composer gives the user complete control over individual functions and their segmentation, however, the user needs to implement often used primitives in distributed data processing frameworks themselves.

2.3 Applicability of DSU to Distributed Data Processing Frameworks

Unfortunately, these previously mentioned techniques are not feasible candidates for realizing DSU capabilities for distributed data processing frameworks such as Apache Spark with currently available framework APIs. Those techniques and applications require the user to have complete control over every resource of the data processing system. In the case of systems using a data processing framework, the user has no control over system resources. The data processing framework now takes control of system resources and automatically allocates and manages resources. Also, the submitted user application is not directly part of the data processing framework, making it impossible to incorporate the DSU functionality due to the aforementioned limitations due to lack of resource control. A potential solution would be to re-write core parts of the API of the data processing framework of choice. The re-writes would change the core functionalities of the data processing framework to support DSU capabilities.
There have been attempts at such a re-write of a data processing framework. One such example is the JStorm framework developed by Alibaba, which is a partly rewrite of the Apache Storm project. The JStorm framework offers new utilities, which are not present in the original Apache Storm project, which provide DSU like behaviour for parts of the pipeline [4]. For example, the framework offers the ability to dynamically adapt the number of workers and also add and new components to the pipeline. However, the JStorm project is no longer updated and documentation is lacking

2.4 Specific requirements of DSU to Distributed Data Processing Frameworks

In general, a data processing pipeline is a static, fixed structure of operations, with a well-defined environment. The data source, sequence of operations, and eventual goal are all fixed in the pipeline. Making all these static elements dynamic and react to changes causes several challenges. Of course, if the runtime of the application is short enough, restarting the processing pipeline after updating the source code is a feasible solution. However, if applications are executed for a very long time (or even continuously, in the case for streaming data), or their availability is paramount, on-the-fly modification of the processing pipeline is not only useful, but necessary.
Pipelines where the data processing source is continuous (such as streaming data), or where the workload is sufficiently large, thus have similar needs for this dynamic updating. The solutions that are offered apply to both of these processing pipeline configurations.
However, the solutions are constrained by the platform that the data processing pipeline is provided by. In the coming sections, Apache Spark is used as the distributed data processing framework. The use of a third-party implementation provider limits the direct control the user has over the application runtime, and thus limits the venues by which can be searched to provide a solution.

2.5 Update Safety

As with any distributed system, the user needs to ensure that all components of the distributed system work together. This requires some kind of mechanism which ensures that updates posted to the system are valid and do not endanger the running state of the system.

2.6 Dynamic Data Source Switching

Another challenge lies in dynamically switching between data sources. A data source refers to the origin of the data received by the distributed data processing framework. This could either be a database, or perhaps streaming data. The research by Lazovik et al. proposes a solution for dynamically switching between data sources [12].
Dynamic data source switching is useful in a data processing pipeline, as it allows the application not to be restarted in other to use another data source (or if we want combined results from two or more data sources). In an ideal world scenario, on-the-fly data source switching should not require more code than using only a single data source.
Unfortunately, in the case where the data processing pipeline resides within Apache Spark, it is not possible to natively instruct Apache Spark to switch between data sources, because an API providing such functionalities does not exist.
Additionally, dynamic data source switching requires the user to ensure that both data sources are heterogeneous, as the processing platform no longer has the same data source, but still performs the same operations on the data.
This introduces the need for the user to provide some sort of intermediary layer between the data and the Spark instance, in order to allow for this dynamic switching. Lazovik uses the term Data Access Layer (DAL) that facilitates this sort of design. The DAL is responsible for communicating with the data source and is able to indicate which subset of the data is required. In general, database developers provide a so-called driver through which the user can interact with the database. the DAL can use this driver to fulfill its tasks. Furthermore, it can frequently happen that a data source does offer any interface by which we can select subsets of the data (e.g. a text file containing the data). the DAL then needs to take care of filtering all data in order to provide the requested data.
Furthermore, in the case of a fixed pipeline, but using a different data source, the DAL must account for difference in data types and data structures between the different data sources. The DAL thus requires some sort of data transformation elements in order to ensure compatibility and homogeneity of the two different data sources.
Finally, we want the DAL to take data locality into account. Data locality refers to performing operations on data as close as possible to where the data resides in memory (i.e. on the node that also stores that subset of the data source). We thus want the DAL to distribute the data in such a way that the data that each worker node has to process is on that exact node. This prevents data from being moved around from node to node, which slows down the processing speed.
To summarize, the DAL should function in the following concrete ways:

[3] https://openwhisk.apache.org/
[4] https://github.com/alibaba/jstorm

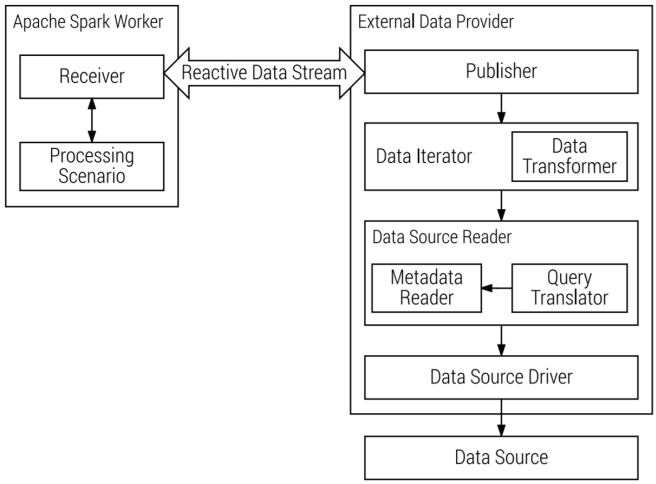


Fig. 1. Architecture of a Spark data processing pipeline using an external data provider for data source switching [12]

- Interact with data source and indicate which subset of data source is required
- Retrieve the selected data (either through queries, or filter the data based on selected criteria)
- Abstract away difference between data formats to present consistent data to the application
- Take into account data locality when providing batches of data

2.6.1 External Data provider

This implementation resorts to using an external data provider, acting as the DAL.
Normally, the data source is very tightly coupled to the data processing pipeline, or it may even be considered a direct element of it. If we decouple the data source, the pipeline no longer has a system that is always reachable in the vacant spot where the data source normally is. The user thus needs to introduce some sort of system that takes on this role. Furthermore, we would still like to guarantee data locality. This would mean that the processing pipeline external data provider and the data source(s) are located on the same node in the cluster, and distributed in the same way as the data processing pipeline and the data source(s).

3 State of the art solutions

The aforementioned limitations show that there is a need for more sophisticated solutions, which allow for more direct control over the distributed data processing pipeline. There are three promising state-of-the-art approaches which will be discussed together with their advantages and disadvantages.

3.1 Dynamic Software Product Lines

A software product line (SPL) is defined by the Software Engineering Institute (SEI) as a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [5]. The key aim of an SPL is to create a software architecture according to which reusable core components can be developed. These core components can then be used to create new products, allowing for more variability whilst decreasing the cost of development time. SPLs have been extended by dynamic software product lines (DSPLs) [9]. In a DSPL, the software that is developed is capable of adapting to fluctuations in user needs and evolving resource constraints. DSPLs have so called *variation points*, which are bound at run time. These *variation points* are initialized based on environmental conditions and are changed when those conditions change. However, the possible values of such *variation points* have to be predefined: an application submitted to the pipeline controller is immutable.
Research using such DSPLs with *variation points* in combination with Apache Storm as a data processing framework has shown the feasibility of such an approach [16].

[5] www.sei.cmu.edu/productlines

3.1.1 Advantages and Disadvantages

However, the limitations are very clear: it is not possible to introduce new values (constants, algorithms, etc.) which can be bound after the data processing pipeline has been started without restarting the pipeline. This means that only those values which are anticipated before the start of the data processing pipeline and thus are included, can be used as values for the *variation points*. Although this approach improves upon a completely non-dynamic approach, allowing for at least some dynamicity, it is still very limited. It is not possible to account for all possible feature values of the *variation points*, thus another more sophisticated approach should be considered. Such an approach should offer the ability to update the *variation points* of an already running data processing pipeline.

3.2 Google Cloud Dataflow

Google Cloud Dataflow is a managed pipeline solution which offers DSU functionality only for streaming jobs when powered by the Apache Beam SDK [3]. The functionality is achieved by replacing the existing streaming job with a new job, which runs the updated application. It is possible both to scale the streaming jobs on the fly, by adding workers and to update the functionality of the element in the streaming processing job. Due to processing pipeline being managed by Google's cloud environment, downtime to a lack of resources whilst spinning a replacement processing job is limited. Also, Google Cloud Dataflow makes several statements about updating a streaming processing job:

(i) In-flight data will still be processed by the transforms in your new processing job

(ii) There is no guarantee that new processing job elements take effect directly, depending on whether the in-flight is buffered

(iii) The new processing job needs to pass a job compatibility check to prevent breaking compatibility between the new and the old processing job

3.2.1 Advantages and Disadvantages

The Google Cloud Dataflow solution offers the ability to update your streaming processing job in a managed cloud environment. The cloud environment solves scalability issues and provides clear guarantees when it comes to updating the streaming processing job. There are also disadvantages which come with Google's solution. it is not possible to dynamically switch data sources, since the data is buffered. Also, is not possible to choose any other distributed data processing framework than Apache Beam.

3.3 Spark-dynamic

Another solution to the problem of applicability of DSU functionality to popular data processing frameworks was proposed by Lazovik et al. [12]. The research developed a proof-of-concept framework called *spark-dynamic*, which is build on the data processing framework Apache Spark [19]. The proposed proof-of-concept framework offers the DSU functionality where parameters and algorithms in various steps of the data processing pipeline can be adapted without restarting the pipeline itself. Lazovik et al. split the problem of DSU with respect to distributed data processing frameworks into two subproblems: *on-the-fly switching between data sources* and *dynamic change of functionality in processing elements within one processing pipeline*.

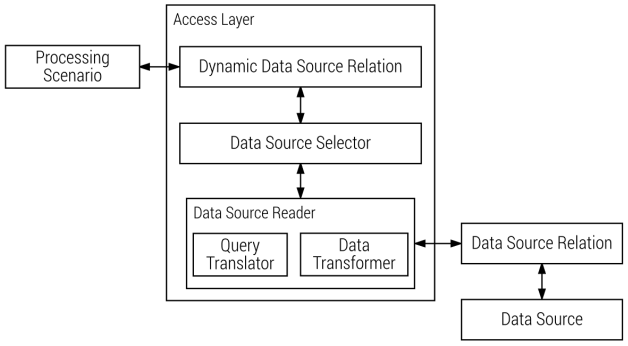


Fig. 2. Architecture of *dynamic-spark* Spark extension for data source switching [12]

3.3.1 On-the-fly Switching between Data Sources

Two methods for switching in an on-the-fly manner between data sources have been proposed. The first method that was proposed involves a custom Apache Spark extension. By creating a custom spark extension using the API from SparkSQL which acts as an extra layer between the data processing pipeline and the data sources. The resulting architecture is shown in Fig. 2. This solution is specific to the Apache Spark distributed data processing framework, due to the use of APIs specific to Apache Spark.
A more generic approach has also been developed in the form of the data access layer (DAL) which has been mentioned in the background section already. This approach showed to lead to a viable approach in which dynamically switching between data sources was possible.

3.3.2 Dynamic Change of Functionality in Processing Elements within one Processing Pipeline

A method for dynamically changing parameters and algorithms within elements of one processing pipeline was also proposed. In this approach Lazovik et al. created wrapper functions for the core functions of Spark. These wrapper functions then make use of a polling based approach, in which a REST update server is polled for updates. If an update is found using polling, the update mechanism is triggered.

3.3.3 Advantages and Disadvantages

The *dynamic-spark* prototype provides true DSU functionalities, making it possible to dynamically switch between data sources and dynamically update elements of the processing pipeline. However, there are still some limitations which should be given attention to. For example, switching between heterogeneous data sources remains a problem. For this, some kind of query translator needs to be designed, which allows the DAL layer to handle the heterogeneity of the various data formats and interfaces associated with different data sources. Also, there is a slight cost in resources (around 10% max.) due to overhead. The authors also mention various other shortcomings related to the dynamic updates: version synchronization, error handling and dynamic execution plan changes. For example, in the current state the prototype cannot prevent that different workers in the distributed data processing pipeline are on different versions. Such a state could occur due to updates not propagating correctly. Also, the current way of error handling is lacking: an error due to a dynamic software update will stop the pipeline. However, the described work has clearly shown the feasibility of an implementation which realizes real DSU functionalities in a distributed data processing pipeline.

4 Points of Caution

Companies (users) want to limit the effects of updates. IBM introduces an interesting perspective with regards to performing updates: it is transformed into a cost-benefit analysis [11]. Gartner estimates the cost of downtime to be around $5,600.00 per minute, or $366,000.00 per hour in general [1]. One could see why preventing downtime is of utmost importance to companies.
Also introduced is the ripple effect, which has stronger ties to software that serves multiple people. For example, a web-based store. In this example, downtime in this sort of software leads to people not being able to place their orders, which leads to lower customer satisfaction, which leads to decreased likelihood of orders being placed in the future. The effect caused by downtime of the application ”ripples” through the various users and eventually causes devastating consequences.

(i) Updates can cause downtime

(ii) Updates can cause fallout

(iii) Updates may be retracted

(iv) Updates may be partially applied

IBM proposes a number of steps one might undertake in order to mitigate the impact of performing updates on a running system.

4.1 Take a holistic view of IT technical support strategy

In order to deal with fallout of dynamic updates, or performing updates in general, it is advisable to have either on-site support or remote support. Furthermore, it proves beneficial to focus attention on incident management and preventative maintenance (either through a supporting team or allocating company time). Additionally, it is imperative that the software is able to produce detailed error reports whenever the system experiences unforeseen circumstances.

4.2 Conduct a thorough assessment of your current IT support structure

Solving the problem ad hoc can be good for unconventional problems, but will most probably yield unconventional solutions. These solutions are not sustainable in the long-term, and users should try to undertake preventative steps in order not to resort to ad hoc solutions. One can also prepare for downtime in a reactive manner: a keystone in this approach is providing a single point of contact for end users. Additionally, Service-Level Agreements can be implemented and agreed upon in order to protect both the user and the provider (e.g., promises over percentage of uptime of your service).
Additionally, it can be useful for companies to reflect on the impact downtime has on their customers and on themselves, as it could be that this changes based on the service the company provides, but also the number of users of the service. In line with this evaluation is considering which parts of your service are mission-critical, i.e. what causes a company the most pain to resolve.

5 Conclusion

We have discussed what data processing pipelines are, why they are useful, and how they work. We discussed particular frameworks such as Apache Spark that provide these data processing pipelines in detail. However, the drawback of frameworks like Apache Spark is that the user hands control of resources over to the framework, which limits the venues over which the user can dynamically update the pipeline without making modifications to the framework source code. Careful thought should be given to making modifications to the core functionalities of the data processing frameworks. Is it really the job of the data processing framework to allow for DSU or should they just provide the core functionality of processing data. In the current landscape, where multiple data processing frameworks exist, a layer which wraps around the existing frameworks and provides DSU functionalities in such a way, would be ideal. No adaptations of core functionalities of the data processing frameworks are required, whilst providing the new DSU functionalities. However, no such truly general approach exists yet.
We evaluated current software implementations that provide dynamic

software updating in the context of distributed data processing frameworks. Most of the implementations providing dynamic software updating functionality are not applicable in the domain of distributed computing in combination with the chosen data processing pipeline. Unfortunately, if users do not use dynamic software updating with data processing pipelines, this usually incurs costs. The user can gracefully shutdown the pipeline after which an updated pipeline can be started. However, this means that there will be a period of downtime, which might lead to high costs. Alternatively, the user can run another updated pipeline in parallel and switch off the old pipeline once the updated pipeline is fully up and running. However, this might be complex and costly, since a complete replica of the infrastructure is needed for running the updated pipeline. The provided solutions for data source switching, such as using a Data Access Layer (DAL) or an external data provider are implementations that can work regardless of the data processing pipeline framework of choice. We evaluated Dynamic Software Product Lines (DSPL), which introduce variation points in the software which can be changed based on environmental conditions. Additionally, we discuss *spark-dynamic*, a proof-of-concept implementation framework which allows for some facets of DSU: on-the-fly switching between data sources and dynamic change of functionality in processing elements within one processing pipeline. Finally, we discuss the impact (faulty) updates can have on a running system. We provide concrete steps in mitigating the downtime that can resolve from performing system updates, and steps towards preventing any further fallout. We discuss the steps towards developing a roadmap for dealing with updating a running system.

REFERENCES

[1] The cost of downtime. https://blogs.gartner.com/andrew-lerner/2014/07/16/the-cost-of-downtime/. Accessed : 22-02-2019.

[2] Streaming programming guide: upgrading application code. https://spark.apache.org/docs/latest/streaming-programming-guide.htmlupgrading-application-code. Accessed : 20-02-2019.

[3] T. Akidau, R. Bradshaw, C. Chambers, S. Chernyak, R. J. Fernández-Moctezuma, R. Lax, S. McVeety, D. Mills, F. Perry, E. Schmidt, et al. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment*, 8(12):1792–1803, 2015.

[4] M. Bhandarkar. Mapreduce programming with apache hadoop. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, pages 1–1. IEEE, 2010.

[5] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[6] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew. Polus: A powerful live updating system. In *29th International Conference on Software Engineering (ICSE'07)*, pages 271–281. IEEE, 2007.

[7] M. Endler and J. Wei. *International Workshop on Configurable Distributed Systems*. 1992.

[8] M. Fu, A. Agrawal, A. Floratou, B. Graham, A. Jorgensen, M. Li, N. Lu, K. Ramasamy, S. Rao, and C. Wang. Twitter heron: Towards extensible streaming engines. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1165–1172. IEEE, 2017.

[9] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, April 2008.

[10] G. Hjalmtysson and R. Gray. Dynamic c++ classes-a lightweight mechanism to update code in a running program.

[11] IBM Global Technology Services. https://www-01.ibm.com/common/ssi/cgi-bin/ssialias?subtype=WH&infotype=SA&appname=GTSE_MT_MT_USEN&htmlfid=MTW03011USEN&attachment=MTW03011USEN.PDF, 2014. Accessed: 20-02-2019.

[12] E. Lazovik, M. Medema, T. Albers, E. Langius, and A. Lazovik. Runtime modifications of spark data processing pipelines. In *2017 International Conference on Cloud and Autonomic Computing (ICCAC)*, pages 34–45. IEEE, 2017.

[13] I. Neamtiu, J. Bardin, M.Uddin, D.-Y. Lin, and P.Bhattacharya. Improving cloud availability with on-the-fly schema updates. 2013.

[14] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for c. *SIGPLAN Not.*, 41(6):72–83, June 2006.

[15] R. P. Cook and I. Lee. Dymos: A dynamic modification system. *ACM SIGPLAN Notices*, 18:202–202, 08 1983.

[16] C. Qin and H. Eichelberger. Impact-minimizing runtime switching of distributed stream processing algorithms. In *EDBT/ICDT Workshops*, 2016.

[17] A. Ramaswamy, S. Bratus, S. w. Smith, and E. Locasto. Katana: a hot patching framework for elf executable. https://www.cs.dartmouth.edu/~sws/pubs/rbls10.pdf.

[18] S. N. Srirama, P. Jakovits, and E. Vainikko. Adapting scientific computing problems to clouds using mapreduce. *Future Generation Computer Systems*, 28(1):184 – 192, 2012.

[19] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

university of groningen / faculty of science and engineering / computing science