FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Deutscher Titel

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Bernhard Gößwein

Matrikelnummer 01026884

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Mitwirkung: Dr. Tomasz Miksa

Wien, 1. Jänner 2001

Bernhard Gößwein

Andreas Rauber

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Designing a Framework gaining Repeatability for the OpenEO platform

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Bernhard Gößwein
Registration Number 01026884

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Andreas Rauber
Assistance: Dr. Tomasz Miksa

Vienna, 1st January, 2001

Bernhard Gößwein	Andreas Rauber

Erklärung zur Verfassung der Arbeit

Bernhard Gößwein
Vorderer Ödhof 1, 3062 Kirchstetten

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Jänner 2001

Bernhard Gößwein

Danksagung

Ihr Text hier.

Acknowledgements

Enter your text here.

Kurzfassung

Ihr Text hier.

Abstract

Enter your text here.

Contents

CHAPTER 1

Introduction

1.1 Problem Description

Over the last decades remote sensing agencies have increased the variations of data processing and therefore the amount of resulting data. To preserve the data for further usage in the future it is necessary to have citable data and processes on the data to ensure repeatability in a long-term.[?] Already most of the the data used in earth observation sciences are retrieved or provided via Service Oriented Architecture (SOA) interfaces. Provider like Google Earth Engine and EODC provide an Web API for retrieving and processing data. Due to a different range of functionality and a difference between the endpoints of the providers it is hard to create a workflow for more than one provider. The OpenEO project has the goal to be an abstraction layer above different EO data providers. The underlying structure consists of three parts:

- Client Module : Is written in the program language of the user and transfers the users commands to the backends.
- Core Module: A standard on how the communication should take place between client and backend.
- Backend Module: The provider of the data and the services, which gets the instructions from the clients and returns the results.

Further information on the software architecture of the project is defined in the project proposal ([?]).Until now there is no consideration of repeatability verification of workflows for users in the OpenEO architecture.Generalised layers have the opportunity to be implemented in a way that makes processes and data scientifically verifiable and reproducible, because it handles data and processes on the data in a standardised way on different providers. Even though the range of functionality and the API endpoints are well defined in the OpenEO coreAPI the contributing content providers (OpenEO backends) will have different underlying software types and versions. The underlying technology of an OpenEO backend will also change over time and can lead to different results on the same workflow executions. Consider the following: A scientist runs an experiment using OpenEO as his research tool and gets results. The same scientist runs the same experiment with the same input some months later and gets slightly different results. The question occurs, why are the results different? Has the used data changed, has the user accidentally submitted different code or has some underlying software inside the backend provider changed. Adding a possibility for the users of OpenEO to gain this information is an important feature for the scientific community. The aim of this thesis is to provide a possibility for users of OpenEO to verify and validate a job re-execution on different underlying technologies of an OpenEO backend provider.[?]

Read through and improve above text.

1.2 Aim of this Work

The expected outcome of this thesis is to discover and develop a possible framework for providing repeatability in the OpenEO project. This enables users to re-execute workflows and validate the results, so that differences on the process or data are accessible for the users. To achieve this goal a model for repeatability within the project has to be discovered and implemented to evaluate the ability of the model. The model shall then conclude recommendations for the OpenEO project on how to improve re-execution validation for the user and how it can be achieved. Therefore the following research questions can be formulated:

- **How can an OpenEO job re-executed be applied like the initial execution?**
  - How can the used data be identified after the initial execution?
  - How can the used software of the initial execution be reproduced?
  - What data has to be captured when?
  - How can the result of a re-execution in future software versions be verified?
- **How can the equality of the OpenEO job re-execution results be validated?**
  - What are the validation requirements?
  - How can the data be compared?
  - How can the re-execution be validated after changes of the OpenEO backend environment?
  - How can differences in the environment between the executions be discovered?

Read through above text and improve.

Add description of use cases (see summary).

CHAPTER 2

Related Work

Currently, there exists no concrete solution to add the ability of repeatability to the OpenEO project. However there are concepts of adding repeatability in computer science.

2.1 Reproducibility

2.1.1 eScience

The eScience has the potential to enable a boost in scientific discovery by providing approaches to make digital data and workflows citable. In [?] is a common way of reaching this goal formulated. It describes an approach to look at whole research processes, other than only data citation by introducing Process Management Plans. The capturing, verification and validation of the needed data for a computational process is also demonstrated within the paper.[?]

2.1.2 Data Citation

Since the earth observation community use a high amount of satellite data and also within the OpenEO project a lot of big data sets are being used, there needs to be a solution to cite the used data in a workflow. The Research Data Alliance (RDA) working group on data citation provides a 14 step recommendation of data citation. It contains solutions not only for static, but also for dynamic data, so data that changes over time. Using the guideline for data citations from the RDA makes the data scientifically citable. [?] In earth science there is also a strategy of ESA and NASA to achieve a content standard for data preservation.[?]

2.1.3 Provenance Data

The re-execution of an OpenEO workflow not only needs data citation, but also the information of how the workflow was executed. Therefore provenance data has to be captured.[?]There are already several provenance models defined in the scientific community. One of the existing models is the PROV model, which was published in 2013 by the World Wide Web Consortium Provenance Working Group and consists of recommendations and guidelines for provenance data.[?] Another model is the VFramework, designed for the purpose of redeployment including the verification of a re-execution of the same workflow. [?]

2.2 Related Work in the EO Sciences

2.3 Related Tools

Read through above text and improve.

2.3.1 ReproZip

Add section about ReproZip.

2.3.2 Smartcontainer

Add section about Smartcontainer.

2.3.3 Reproducibility in computational geoscience

2.4 OpenEO Details

The OpenEO project contains of three modules, the client modules written in the programming language of the users, the back end drivers that makes it possible for every backend to understand the calls from the clients and the core API that specifies how the communication should take place. So the core API is a standard that the back end providers accepted to implement on their systems. The back end drivers are the translation of the client calls to the back end specific API. This architecture decouples the clients from the back ends so that every client can connect to every back end that applies to the OpenEO core API standard see figure

The communication is specified as an OpenAPI(TODO: Quelle) description and consists of all RESTful services that can be called at the back ends. Most of the calls are only to receive information about the processes and the data of the specific back end, but there are also endpoints to apply job executions specified by the user. In this thesis the focus lies on the job execution, because this is the core feature of the OpenEO project and needs to be reproducible. In the next chapter the job execution process is described in more detail.

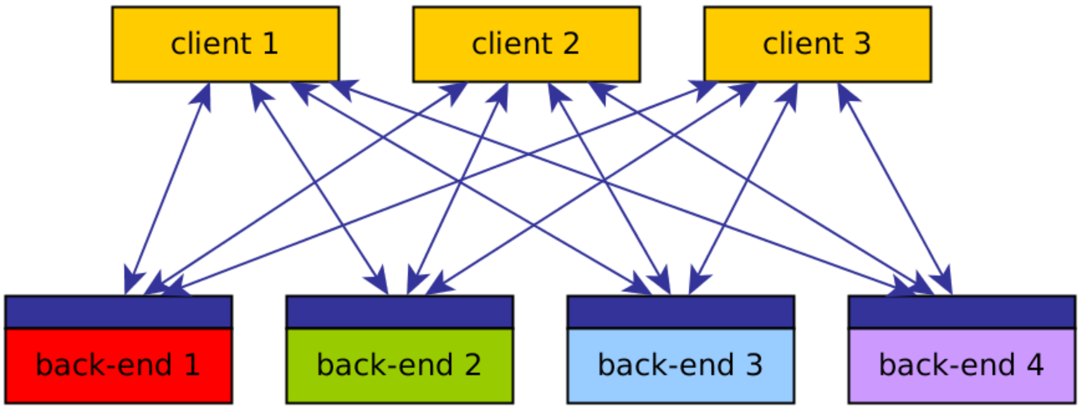


Figure 2.1: Overview of the OpenEO architecture. (from openeo.org)

2.4.1 Job Execution

The job execution workflow in OpenEO starts at a client application that let the user define what has to be processed in the specific programming language. The main part of the job execution is based on the description of what the back end needs to do with what data. Therefore OpenEO introduces the process graph, which consists of a tree structure describing the processes with their data and the input data identifier, which is back end specific. The process graph is in a JSON format and gets generated by the clients in the background without the users seeing it directly. In figure

The back ends interpret the process graph from inside out. Inside figure TODO the element in the middle defines the input data identifier in the ïmagerybblock, with the p̈roduct_id:. In this case the s̈2a_prd_msil1cïs the chosen input data identifier. Now the back end goes one step up in the hierarchy of the process graph and calls the process f̈ilter_bboxẅith the parameters l̈eft;̈ r̈ightänd so on using the input data. The output data of the previous process is also the input data of the next process. It is possible that the input of a process is a whole new process graph. In figure

Add Diagramm of Process Graph

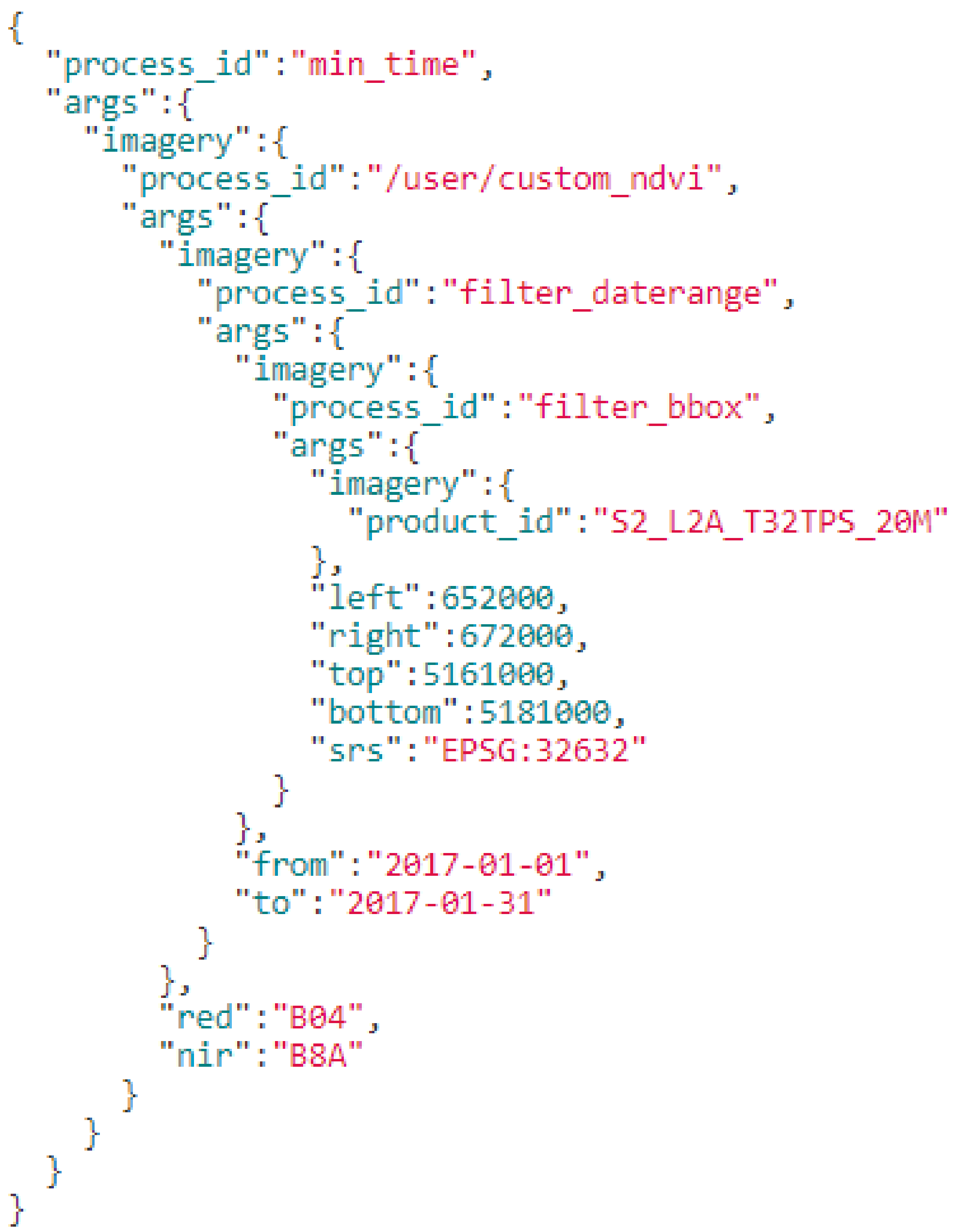
There are two different kinds of process executions depending on the back ends capabilities, synchronous and asynchronous calls. Synchronous calls are directly executed after they got received from the back end and the user that sends the process graph has to wait until the job is finished. For example on the python client the program waits after sending the process graph to the back end until the back end returns the result and the results are directly returned to the user. On the asynchronous call the user sends the process graph, but it does not get executed until the user starts the execution on the back end through an additional endpoint. When the processing is finished the user can download the result at another endpoint of the back end. For the asynchronous calls there is also the possibility to subscribe to a notification system on the backend, so that the user gets

notified when the job execution finishes. The processes are defined at the OpenEO core API and therefore independent from the back end they get called, other than the data identifier, which is different for every back end.
The previous example shows a process graph that only uses predefined processes and data. Within the OpenEO project there is the possibility to define individual processes and execute them on the back end. In the project they are called “user defined functions” and are at the writing of this thesis not well defined, but are basically code written by the OpenEO user that gets sent to the back end and executed at a secure environment. The user can define processes and can run them with the data provided at the back end, using the infrastructure of the back end. Every back end has to individually define what the restrictions on user defined functions are. There is also the possibility to upload files to the back end that can be used within the process graph. The user can upload the file through an explicit endpoint of the back end. After the upload finished, the file can be referenced inside the process graph.

2.4.2 Back end Overview

Even though the back ends implement the OpenEO core API standard, they are still very diverse behind the abstraction layer. Some back ends has already an API, where the OpenEO calls have to be adapted to. There are 7 partners within the OpenEO project that are implementing a back end driver. The back ends have to manage the translation of the process graph to the actual code that executes the defined process chain. Also the billing of the users is completely different on every back end.

Add section about EO reproducibility.



```
{
  "process_id":"min_time",
  "args":{
    "imagery":{
      "process_id":"/user/custom_ndvi",
      "args":{
        "imagery":{
          "process_id":"filter_daterange",
          "args":{
            "imagery":{
              "process_id":"filter_bbox",
              "args":{
                "imagery":{
                  "product_id":"S2_L2A_T32TPS_20M"
                },
                "left":652000,
                "right":672000,
                "top":5161000,
                "bottom":5181000,
                "srs":"EPSG:32632"
              }
            },
            "from":"2017-01-01",
            "to":"2017-01-31"
          }
        },
        "red":"B04",
        "nir":"B8A"
      }
    }
  }
}
```

Figure 2.2: Process graph example (from Github Spec)

CHAPTER 3

Design

In this chapter the design of the provenance capturing gets described. First the architecture of the OpenEO project gets described further. Especially the job execution path throughout the parts of OpenEO gets described, because it is the key element for the context capturing. Therefore the back end, where the jobs get executed get described in more detail. After that the context model gets presented.

In this section the concept of the provenance capturing is described. The aim of the capturing is to be easy to implement on the OpenEO back ends, but also powerful enough to make the reproduction of the workflows possible. The whole capturing process can be structured in three thematic parts.

1. **Back end provenance**
   This part describes the provenance of the back end, that is not depending on a job execution. There has to be an automated process running on the process machine to capture every change that influences the execution of OpenEO jobs. A description of more detail can be viewed on chapter

2. **Job dependent provenance**
   This part contains the provenance of a job execution. It consists of the workflow context and the data related to one specific job. This in combination of the back end provenance is the full provenance data to make an OpenEO job theoretically re-executable.

3. **User information**
   This part consists of all possibilities that provides users with information they want to know about the provenance of the back end and the provenance of the job execution. The aim is to provide users with information that makes them easier to choose between back ends. On the other hand the user shall be able to retrieve the

provenance data of the job in a way that the back end has no security risks. This part is not directly part of the context model, but of the implementation on the user interface.

3.1 Back end provenance

The scope of this part of the context model is to get the static environment of where the job execution at the back end takes place. It contains the provenance data that is independent from a job execution, so does not change regardless of how much jobs were processed. It can only change from inside of the back ends by their maintainers. In the OpenEO project there are a great variety of different back end providers with very different setups, so the challenge of this part is to make it as simple and generic as possible. The data captured in this part of the context model is not meant to be shown to the user directly, because of security issues. The following data gets suggested to be the minimal set of static provenance data that has to be captured.

1. **Github Repository**
   Since the project is an open source project and every back end has a github repository were at least the basic setup is stored. The aim of this strategy is to get other back ends with similar settings to reuse the already working setups of the running back ends. Therefore, at least during the project runtime, every back end provider has a github repository were the code is publically available. This information is added to the back end provenance by saving the git repository url, the used branch, the used commit and local changes to the repository.

2. **Used Folders**
   The back end will most probably not only use the github repository for the processing, so that other folders are involved that stores config files or additional code. This information is also crucial for the job execution and therefore needs to be stored. Changes of this directory have to be detected and stored in the context model. Temporary written folders have to be excluded to the capturing to prevent detecting changes that are not important.

3. **OS and packages**
   The operating system can also have an impact on the processing and so it has to be stored to the context model too. Not only the information about the operating system, but also the installed packages have to be stored in the context model. In a linux based system it is sufficient to store the location where the packages get installed (e.g. /etc). If the whole processing is done in a virtual container the operating system of that container shall be stored. If the whole processing is done in a docker container, the docker description file has to be saved in addition. But only if the docker container description does not change on different input process graphs, otherwise it has to be added to the "job dependent provenance".

4. **Core API Version**
   The back ends core api version is the version of the OpenEO core API version it is using. In the productive usage of the OpenEO project it can happen, that the versions of the clients and the back ends differ, so that the behaviour might not compatible. So the back end server configuration is depending on the core API version of OpenEO, hence it shall be added to the context model.

5. **Back End Version**
   The captured data described in the previous sections will result in a back end version. The back end version shall be an identifier (e.g. a number) that gets updated on every change of the backend considering the provenance data described above.

   To provide long term stability of the capturing, there shall be a tool to automatically capture the provenance data to the context model. Every change on any of the previously described context data has to be detected automatically and have to result in a new back end version. This can be done by a standalone capturing tool that does not need to have detailed insights into the back end.

3.2 Job dependent provenance

In this section the job dependent provenance of the context model gets described. The data captured is tied to an specific job execution, so for every job execution a new context model gets created. The structure of the capturing is using the defined process graph structure of OpenEO. The process graph is already a description of the processes that run at the back end. So sending the same process graph to the same back end again shall result in the same outcome. To assure that the process graph results in the same way of processing the processing itself gets captured. After receiving the job the back end will transform the process graph into code that actually does the processing. Figure

The single parts of the capturing are described more in detail in the following sections.

3.2.1 Input Data

The input data of the processing is crucial for the outcome of the process graph. Even though the process graph already uses an identifier for the input data that has to be unique for the specific back end, changes to this data might not result in a new identifier. So jobs called later in time might use another version of the input data than the previous jobs. To prevent this the input data has to be persisted according to the 14 steps of data provenance defined by the RDA[1] described in the related work section. Every back end is responsible for the data versioning, for this context model it is assumed that the back ends preserve the data properly. Nevertheless the current core API version (v0.0.2) is not capable of letting the user choose, which version of data the user want to use. Hence there has to be an addition to the process graph so that product identifier shall have an additional timestamp. The meaning of the timestamp is to tell the back end that

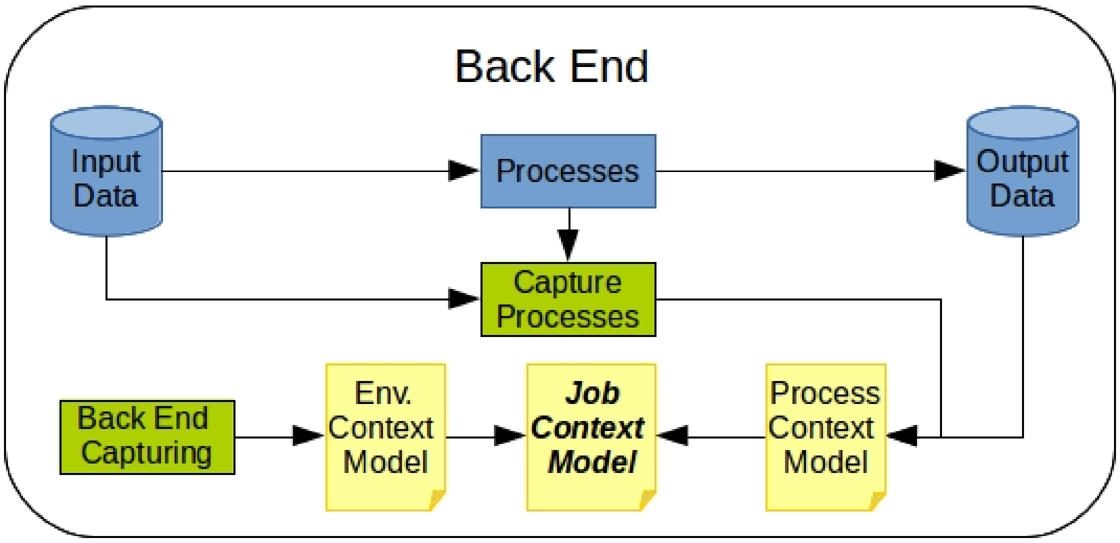


Figure 3.1: Design Overview of capturing the job context model.

the user wants to use the data at the time of the timestamp. This gives the back ends the possibility to design their own data versioning. They just have to remember, which version of the data was available on what date and time. Not only the product used for the process graph is input data but also files uploaded by the user are included at the input data. As long as the file name is accessed at the specific process graph the file gets added to the provenance of the job execution.

To guarantee that the data has not changed even though the version has not changed it is considered that a hash of the input data is added to the provenance data. The earth observation data stored at the OpenEO back ends can be very big and therefore it is recommended that the back end provider already hash the input data after every change on it so that the performance of the job execution is not threatened. This also applies to the uploaded files of the user, where the hash can be done already after the uploading. In OpenEO files that get uploaded with the same filename are automatically overwritten so that it is especially important to take the hash of this files for the context model. Otherwise it is not possible to replicate differences on a job execution.

3.2.2 Process Data

In this section the capturing of the process itself gets described. As mentioned above the process is described at the process graph, but to be certain that the same process id results in the same code, the code has to be captured. So the code running the processing has to be captured and has to result in a signature that can be compared to other executions. An example of doing so is to hash the entire resulting code. The way of executing a specific process graph is not only related to the code running it, but also by the dependencies of the code. So the environment of the code and information about the code environment has to be added to the context model. If there are any docker container, where the job is running that is not independent on the process graph has to be captured. The programming language, its version and its used packages has also to be captured and added to the job context model. To every process executed also the start time and end time has to be captured as a time stamp.

3.2.3 Output Data

The output data of the processing has to be captured as well. In the OpenEO project the results can be rather big, that is why a simple generated identifier of the output result is enough in this context model. An example of this identifier would be a hash over the resulting data. The aim of the output data capturing is not to add the possibility to find differences between results, but to be able to see that the results are different. So for this context model a simple solution is sufficient.

3.2.4 Benchmarking

So far only the capturing of a single process, including input and output data, are described, but not how the whole process graph shall be captured. The basic idea of the job dependent provenance capturing is to capture the input data of the process graph, the whole process graph and the output of the process graph, described like in the sections above. This is a common basis the back end providers can agree on and is not affecting the back end providers implementations much. It is also capable of giving the OpenEO user a simple overview of how the results are generated and what are differences between different job executions. To make the comparison of different execution behaviors more informative, the granularity of the capturing can be improved. So that is how the

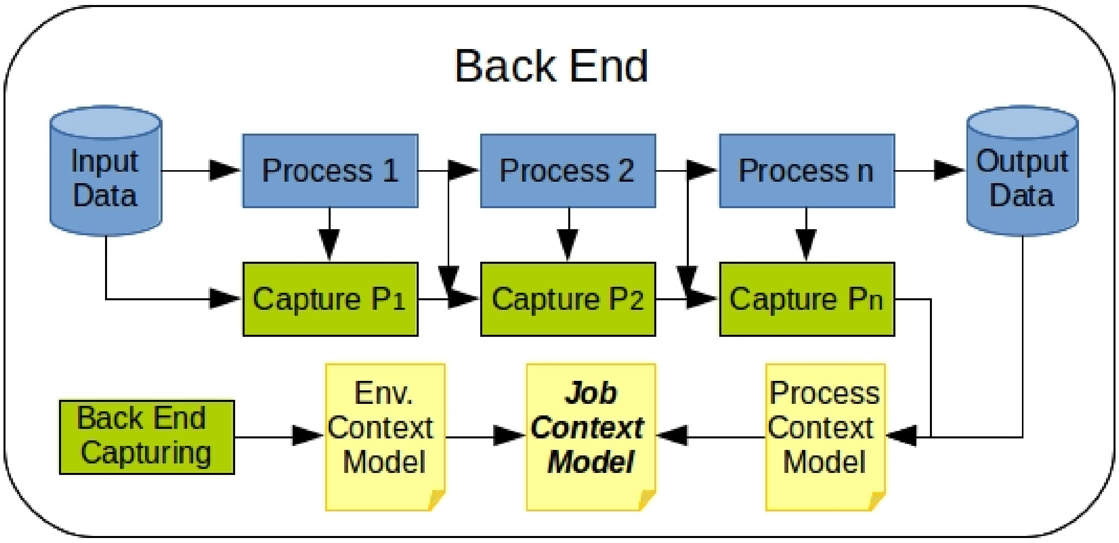


Figure 3.2: Design Overview of capturing the job context model.

benchmarking context model gets introduced. The idea is that not only the input data the whole process and the output data gets captured, but also every data in between of every process in the process graph. So for every process in the process graph the input, output data gets captured and the code executing the process. This makes it easier for users to see were the execution actually changed. But it also comes with higher implementation costs at the back end side. It also will affect the performance of the execution more than the common context model. The possibility of the implementation of such a granularity is also highly dependent on the back end implementation. Not every back end might be able to implement this into the back end due to execution optimization where the execution and results of the single processes in the process graph are not distinctable. Nevertheless in theory the granularity does not have to stop there, the data and code could be captured for every line in the code. This flexibility of capturing makes it also a context model that can be adapted to different use cases in the future.

3.3 User Information

In this section the information for the users get described. The capturing described in the previous sections consists of a lot of information about the back ends that shall not be passed to the users in detail. On the other hand the users need to be able to see the differences of job executions and get environment information about the backends. Therefore there has to be a filter on which data can be shown to the user and what not. Every captured information is also not necessary interesting for the users. Data that is not secure to the user has to be defined by the different back ends themselves. The OpenEO project has a very diverse kinds of back ends and every one has a unique company security guideline. The important thing is that all of the information has to be captured and taken into account for the comparisons and they have to be open to the user, but anonymously if needed. The information has to be able to be provided to the user. Therefore there have to be some new additions to to the core api specification to do so. So there has to be a new endpoint for the users to get information about the back end job independently. Therefore for this thesis the endpoint "GET /version" gets introduced, where the context model of the back end gets provided. There needs to be an endpoint to retrieve the job dependent provenance data, which can be added to the already existing "GET /jobs/<job_id>" endpoint, which is used to get the current status information of the job. The endpoint also includes the back end provenance of the back end during the execution of the job. There shall be an additional endpoint for comparing two different jobs by their captured context model. For this purpose the endpoint "POST /jobs/<job_id>/diff" gets introduced, were the job ids that the user wants to compare with are submitted by the POST body.

3.4 Overview of OpenEO Suggestions

In this section the changes described in this chapter get summarized and assigned to the parts of OpenEO responsible for the tasks.

3.4.1 OpenEO back end

The openeo back ends need to implement the job dependent capturing as described in the section TODO and implemented in section TODO. The job dependent capturing is recommended to be implemented in the same programming language as the back end provider uses. The advantage of this is that the usage of the programming language is maintained anyway by the back ends to preserve the functionality of the processes and if the back end switches to a different location the setup of the capturing tool needs no further effort. The back end also need to run the job independent capturing described in section TODO. This has to be run in a cron job or other automation tools so that the changes get notices by the tool. In section TODO there is also an implemented tool doing so. In addition the backends need to implement the endpoints changes and additions of the OpenEO core API described in the next section.

3.4.2 OpenEO core API

The core API needs to add the endpoint for the version retrieval (e.g. a /version) end point. It is also possible to extend an existing endpoint for this purpose. There needs also an additional endpoint for comparing two different jobs. The process graph shall be extended by a timestamp to the product element so that it is possible to access deprecated data on the back ends.

3.4.3 OpenEO Client

The OpenEO clients need to implement the changes described above in the core API additions. In the OpenEO client the additional retrieved data needs to be visualized for a better user understanding.

Read through above text and improve.

Add section about noworkflow

CHAPTER 4

Implementation

Enter your text here.

CHAPTER 5

Evaluation

5.1 Use Cases

5.2 Using Python Client

CHAPTER 6

Conclusion and future Work

Enter your text here.

List of Figures

List of Tables