TU WIEN Informatics

Provenance-Erfassung in Python-Umgebungen

Herstellen von PROV-O-Konformität für YesWorkflow und noWorkflow und Beurteilung des resultierenden Nützlichkeitszuwachses

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Raffael Foidl
Matrikelnummer 11775820

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Ao.univ.Prof. Dr. Andreas Rauber

Wien, 31. März 2020

Raffael Foidl	Andreas Rauber

TU WIEN Informatics

Provenance Capturing for Python Environments

Achieving PROV-O Compliance for YesWorkflow and noWorkflow and assessing the resulting Increase in Utility

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Raffael Foidl
Registration Number 11775820

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.univ.Prof. Dr. Andreas Rauber

Vienna, 31st March, 2020

Raffael Foidl | Andreas Rauber

Erklärung zur Verfassung der Arbeit

Raffael Foidl

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, 31. März 2020

Raffael Foidl

Kurzfassung

Wissenschaftliche Experimente gehen überlicherweise mit zahlreichen Rechenschritten einher, die gewisse Eingabedaten entgegennehmen und für die Erzeugung von Resultaten verantwortlich sind. Solche Arbeitsabläufe werden oft mithilfe von Skripten abgebildet beziehungsweise umgesetzt. Skripte können zwar an jede Anforderung angepasst werden, doch sammeln sie in der Regel keine wertvollen Metadaten wie Experiment-Eingabedaten, Prozessabläufe oder Dateizugriffe – kurz gesagt, Provenance. Diesem Problem wurden bereits mehrfach Forschungsbemühungen gewidmet, woraus Programme entstanden sind, die Provenance für Skripte sammeln. Diese verwenden jedoch nur selten ein gemeinsames Provenance-Format, was die Verarbeitung und den Austausch ihrer Ausgabedaten erschwert. In dieser Bachelorarbeit werden zwei solcher Ansätze – YesWorkflow und noWorkflow – derart erweitert, dass ihre Erzeugnisse konform zum PROV-Standard des World Wide Web Consortiums sind. Das Hauptziel dabei ist es, die Nützlichkeit der Ausgabedaten zu erhöhen, indem Interoperabilität und maschinengestützte Weiterverarbeitung erleichtert werden. Diese Arbeit umreißt, wie die vorgestellten Änderungen für YesWorkflow und noWorkflow umgesetzt wurden und wie aus ihnen Nutzen gezogen werden kann. Dabei werden Möglichkeiten aufgezeigt, wie die ontologische Darstellung der Provenance tiefere Einblicke in die Struktur von Skripten gibt und Informationen über vergangene Testläufe bietet. Hierzu werden RDF-Serialisierungen der PROV-Ontologie herangezogen, um auf sonst nur implizit vorhandene Details schließen zu können. Dies wird mittels eigens konstruierter SPARQL-Abfragen demonstriert.

Abstract

Scientific experiments commonly involve several computational steps that consume certain input data and are responsible for the generation of results. Such workflows are often encoded or implemented in the form of scripts. While scripts can be adapted to every requirement, they lack valuable metadata such as experiment parameters, process flows and file accesses – in short, provenance. There have been research efforts addressing this issue which resulted in tools that collect provenance of scripts. However, they rarely utilize a common provenance format, which makes processing and exchanging their outputs difficult. In this thesis, two of these tools – YesWorkflow and noWorkflow – are extended such that they produce results that are compliant to the World Wide Web Consortium's PROV standard. The main goal is to increase the utility of their output by facilitating interoperability and machine-aided processing. This work outlines how the proposed modifications to YesWorkflow and noWorkflow were implemented and how they can be leveraged. Moreover, capitalizing on the ontological representation of their provenance, possibilities to gain deeper insight into the scripts' structure and execution details are highlighted. To this end, RDF serializations of the PROV ontology are used to infer otherwise only implicitly available information. This is demonstrated with the help of specifically constructed SPARQL queries.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Conducting experiments and analyzing their results is of crucial importance in the scientific community. For this purpose, researchers frequently utilize Workflow Management Systems. They provide infrastructure to define workflows representing experiments such that they can be executed, monitored and subsequently inspected in a controlled environment. [DF08] Other scientists, however, prefer manually conceiving scripts that manage the control and data flow. The technology of choice most often is a programming language that is interpreted at runtime. Reasons might include platform independence and immediate adoption of source code modifications, i. e. for re-executions, no tasks such as recompilation are required.

One key feature of Workflow Management Systems is the collection of metadata that documents – among others – experiment parameters, process flow (e. g. file operations, database accesses, external services) and procedures involved in the generation of (specific) results. These records describe the origin of experimental outcomes as accurately as possible and are generally referred to as *provenance*. [LLCF10] When employing self-written scripts, provenance data is not available, at least not without any further action.

Since Workflow Management Systems do exhibit certain shortcomings, many scientists prefer writing scripts, despite the lack of provenance. Thus, research efforts have been made to address this issue (see Chapter 2). As scripts – unlike Workflow Management Systems – do not necessarily run in similar and controlled environments, there cannot be a single solution for all use cases. The techniques, methods and processes necessary to capture provenance of scripts are highly dependent on the language in question and the tools it provides. The most prominent scripting languages in scientific computing presumably are Python, R and MATLAB. Due to its general-purpose nature and extensive use within the community, this work exclusively focuses on Python.

1.2 Problem Statement

Some substantial drawbacks of Workflow Management Systems are limited portability to other platforms or even systems (not all researchers might be familiar with every Workflow Management System), difficulties when integrating external services and the use of proprietary formats [DF08]. The latter one – the absence of an open, standardized model enabling provenance sharing – is the disadvantage that is primarily relevant in the context of this thesis.

For the purpose of this work, two existing approaches to capturing provenance of Python scripts are examined – YesWorkflow [MSK+15] and noWorkflow [MBC+14]. The first one focuses on *prospective provenance*, i. e. information that can be retrieved by statically analyzing the source files, such as method signatures or function call sequences. The latter one is additionally capable of collecting *retrospective provenance*, which is retrieved by monitoring the execution of a script. This, for instance, yields information on functions' parameters and return values or their execution time. [DF08]

However, the abovementioned tools share the problem of Workflow Management Systems. Their output does not adhere to a standardized provenance format, i. e. it lacks well-defined semantics. This fact is detrimental to the feasibility of exchanging and – preferably automatically – processing it without additional effort.

1.3 Aim of this Work

The objective of this thesis is to extend YesWorkflow and noWorkflow such that they produce output that conforms to a standardized provenance model. The model of choice is the PROV framework [GM13], developed by the World Wide Web Consortium (W3C).

By implementing the intended modifications, a more effective utilization of the provenance being collected is anticipated. This implies gaining deeper insight into script executions and their results than what is possible at the moment. Moreover, capitalizing on the applications and services that implement the standards involved, the general interoperability should be facilitated substantially.

To put it more briefly, the goal of this work is to overcome the issue presented by providing answers to the following questions:

1) How can the output of YesWorkflow and noWorkflow be adapted, so that it is compliant to the W3C PROV standard?

2) To what extent does the W3C PROV-O representation of provenance increase the utility of YesWorkflow and noWorkflow?

The remainder of this thesis is structured as follows:

Chapter 2 deals with previous research and development efforts that have laid the basis for this work, namely Workflow Management Systems, YesWorkflow and noWorkflow.

Chapter 3 provides the theoretical foundations that are relevant for the work at hand.

In Chapter 4, the general strategy and considerations behind approaching the first research question are discussed.

Subsequently, Chapter 5 describes the design and implementation of the proposed extensions to YesWorkflow and noWorkflow in detail.

Chapter 6 addresses the second research question as it demonstrates how YesWorkflow's and noWorkflow's output can be used more effectively owing to their representation in PROV-O.

Finally, Chapter 7 reflects on the findings obtained and mentions issues worth further work and research.

The Appendix contains images and listings that are relevant, but would disturb the flow of reading. Similarly, there are occasional references to GitHub repositories which contain artifacts that would be difficult to typeset within the page boundaries – for instance, source code files as well as input or output data may not be presentable in their entirety. The repositories involved in this thesis are:

1. ProvCaptPyEnvs[1]: Contains an overview of the work done in the course of this thesis, the thesis itself as PDF document, related source files, input and exemplary output data.
2. yw-prototypes[2]: The source code of YesWorkflow, including the modifications hat have been made.
3. noworkflow[3]: The extended source code of noWorkflow.
4. sparql-playground[4]: A slightly customized version of an application for presenting and executing SPARQL queries over arbitrary data sets.

[1] https://github.com/raffaelfoidl/ProvCaptPyEnvs
[2] https://github.com/raffaelfoidl/yw-prototypes
[3] https://github.com/raffaelfoidl/noworkflow
[4] https://github.com/raffaelfoidl/sparql-playground

CHAPTER 2

Related Work

2.1 Workflow Management Systems and Provenance of Scripts

Workflow Management Systems have evolved with the intention of facilitating the conception of scientific experiments. They provide ways to conceptualize, share and even execute data-intensive processes that generally involve several computational steps. Such systems mostly offer a visual programming interface – services, tasks and data items are depicted as shapes that are connected according to the experiment's structure, overall resulting in a directed graph. Since users do not have to be familiar with the implementation of the modules that are used and later invoked by the system, no high proficiency in programming is required. [DF08]

One prominent example of such systems is Taverna, an Apache open source project which was initially created for Bioinformatics [OAF+04], but has been adopted in diverse areas of research. Furthermore, it has recently been re-engineered in order to satisfy latest requirements [MSRO+10]. There are plenty of more notable systems, some of which include Kepler [LAB+06], Triana [TSWH05] and VisTrails [BCC+05].

Workflow Management Systems inherently exhibit advantageous preconditions for provenance capturing. Due to the graphical way in which workflows are designed, prospective provenance comes at very little cost. The need for runtime recorders enabling the collection of retrospective provenance has already been addressed for some projects. For the case of Kepler, one might refer to [ABJF06] or [MBK+09].

As already discussed in the preceding chapter, self-written scripts do not naturally come with such possibilities. In order to bridge this gap, approaches like [BGS08] have been developed. It represents a Python library that provides functions which, when called by a script's author, create provenance records. ProvenanceCurious [HAW13], however, collects prospective and retrospective provenance from a Python script without having

to invoke library functions. Similarly, BURRITO [GS12] registers system calls such as I/O operations in order to link them with input and output parameters from previously extracted prospective provenance. However, it does not facilitate gaining deeper insight into a script's internal operations such as nested/internal function calls, arguments or return values.

This thesis focuses on two approaches to collecting provenance of Python scripts – YesWorkflow and noWorkflow. They are described comprehensively in the following sections.

2.2 YesWorkflow

As already mentioned, YesWorkflow aims at generating *prospective provenance*. Although a more precise definition of this term will be given in Section 3.1, this roughly translates to describing the overall process encompassed by the workflow without any information recorded at runtime. That implies outlining how input parameters are consumed, processed and output parameters generated by activities involved in a workflow.

YesWorkflow intends to uncover the data flow and computational steps in a script on an abstract level. That way, people unfamiliar with the script in question are provided a fast way of grasping the structure of a workflow. This proves especially useful because it removes the necessity of delving deep into the source code just to get a rough idea of how the results are computed from the inputs. [MSK+15]

The tool's authors chose visualization as means of conveying an impression of a workflow. Appendix A illustrates the diagram that YesWorkflow produces when using the small example script as input (see also Section 4.1 for more information on the test scripts used in this thesis). The workflow's name is shown at the top, circles denote its input and output data. Program blocks – computational steps that constitute logical units (e. g. one or more functions in the script) – are depicted as green rectangles. Parameters that are either consumed or generated by such blocks can be observed as yellow, rounded rectangles. The process flow is portrayed with the help of directed edges which connect data flow elements and program blocks (and vice versa).

YesWorkflow extracts provenance from scripts by analyzing special source code comments which contain the information to be presented in the output. To this end, there are several annotations predefined that may be used to denote the components in a diagram like in Appendix A. [MSK+15] The most common annotations are depicted in Table 2.1.

Annotations can be separated in two groups, *standalone* and *qualification*. While the first one contains annotations that hold information independently of others, the latter one consists of those which are not meaningful on their own. For instance, `@desc` provides useful information only if it is accompanied by a primary annotation to which it refers, such as `@begin` or `@param`.

Listing 2.1 shows an excerpt of the small test script's source code in order to exemplify the usage of YesWorkflow annotations.

Annotation	Type	Description
`@begin`	standalone	beginning of a code block; must have corresponding `@end`; nesting is allowed
`@end`	standalone	end of a code block; must have corresponding `@begin`; nesting is allowed
`@in`	standalone	input data of workflow or code block
`@param`	standalone	parameter of a code block, but also `@in` can be used
`@out`	standalone	output data of workflow or code block
`@desc`	qualification	describes a workflow, code block or parameter
`@uri`	qualification	specifies a resource identifier, e. g. a file
`@call`	standalone	denotes a call to another function (which does not need to be modeled as code block)

Table 2.1: Most important YesWorkflow Annotations

```
...
# @begin inst_s.main @desc Processes a list of wind speed measurements
# @in data @desc CSV file with country, code, city, wind_speed as columns
# @out summary_by_country @uri file:by_country.json
# @out summary_overall @uri file:overall.json
def main():
  ...
  # @begin persist_top_ten_by_country
  # @in country_top_ten
  # @out summary_by_country @uri file:by_country.json
  # @call get_classification @desc maps wind speed to Beaufort scale
  persist_top_ten_by_country(top_ten_by_country, by_country_path)
  # @end persist_top_ten_by_country

  # @begin persist_top_ten_overall
  # @param overall_top_ten
  # @out summary_overall @uri file:overall.json
  # @call get_classification
  persist_top_ten_overall(top_ten_overall, overall_path)
  # @end persist_top_ten_overall

  print("Done.")
# @end inst_s.main
 ...
```

Listing 2.1: Exemplary Usage of YesWorkflow Annotations

Note that the annotations could have also been inserted above each function definition instead of at their invocation. YesWorkflow is very customizable in regards to the markup

of source code comments. Subsequently, not only multi-line comments, but also any language other than Python is supported. That is because to the program, only comments are of relevance and not the semantics of the respective programming language.

It is worth mentioning that there have been attempts at collecting retrospective provenance using YesWorkflow without employing runtime recording [MBBL15], which noWorkflow partly relies on. However, this approach exhibits certain limitations. For solutions that link YesWorkflow's prospective provenance to noWorkflow's retrospective provenance, refer to Section 2.4.

2.3 noWorkflow

Unless stated otherwise, the information given in this section is based on [MBC+14].

noWorkflow collects *retrospective provenance* by recording information during the execution of a script. That way, for instance, the runtime of individual functions as well as their input and output parameters are registered.

However, the tool's name is in fact an abbreviation for **n**ot **o**nly **Workflow**, implying that the provenance it records is not only about workflows. Figure 2.1 provides a slightly simplified overview of noWorkflow's architecture.

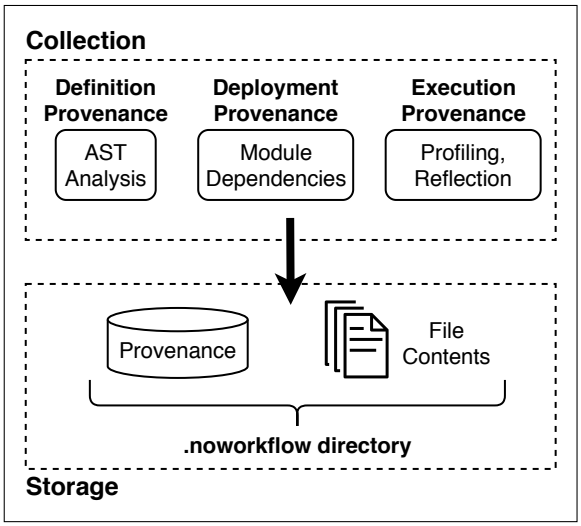

Figure 2.1: Architecture of noWorkflow. Adapted from [MBC+14]

As can be concluded from the graphic, noWorkflow collects and distinguishes between three types of provenance:

Definition Provenance: corresponds to prospective provenance – function definitions, arguments, calls

Deployment Provenance: machine architecture, operating system, environment variables,

	(transitive) module dependencies including their versions
Execution Provenance:	corresponds to retrospective provenance – function activations, execution times, arguments, return values

A function *call* is part of the definition provenance and extracted from a script by statically analyzing its source code. In the context of execution provenance, for one such call there might be numerous function *activations* at runtime.

As opposed to YesWorkflow, noWorkflow does not require any modifications to the scripts' source code. From the user's perspective, the only difference is that scripts are executed using `now run script.py` instead of `python script.py`. Thereafter, noWorkflow runs the script as if it had been invoked ordinarily. Furthermore, it captures the three types of provenance from above by employing techniques such analyzing the script's abstract syntax tree (AST), reflection and profiling. The scope of the provenance collection was chosen to be user-defined functions. Hence, definition provenance only contains functions that have been explicitly declared within the script being executed. Likewise, execution provenance only captures function activations that occur inside such user-defined functions – regardless of whether the activated function is defined within the script or not (e. g. activations of library functions are recorded).

One such noWorkflow-aided script execution is called a *trial*. Trials are uniquely identified by an automatically generated sequence number. Upon execution of the first trial, a folder .noworkflow, which contains two databases, is created in the script's directory. For each trial, the collected provenance is stored in a SQLite database; a separate content database is used for persisting file contents (e. g. source code or files that have been read from or written to). These two databases are linked together via the files' SHA1 hashes.

noWorkflow offers further features not all of which are essential in the context of this thesis. The interested reader might read up on the tool's intricacies regarding script evolution and relationships between trials [PFBM16]. Furthermore, previous research efforts have made it possible to interactively and graphically analyze collected provenance by means of Jupyter notebooks, formerly IPython [PBMF15]. Finally, [PMBF17] provides a fairly detailed rundown of noWorkflow's most relevant features, embedded in examples from real experiments.

2.4 YesWorkflow and noWorkflow combined

On the one hand, YesWorkflow generates user-defined, coarse-grained views of workflows in order to make explicit their structure which is otherwise masked by the complexity of the script's source code. On the other hand, noWorkflow is particularly useful for tracing the lineage of specific results and inspecting the script's behavior based on functions' arguments and return values. The amount of data recorded by noWorkflow, however, might be too excessive – unless an in-depth analysis of the computational processes needs to be conducted. This is what the work of [DBK+15] addresses.

They propose a framework that allows data collected by noWorkflow to be united with the more abstract view of YesWorkflow, while retaining the advantages of prospective provenance. To this end, they leverage the fact that both tools also export their output as Prolog facts. Subsequently, their solution supports visualizing the data – which links retrospective to prospective provenance – as well as querying it using logic programming systems such as DLV.

A similar strategy is pursued by [PDM+16]. They introduce YW*NW, a term representing the joint provenance model between YesWorkflow and noWorkflow, queries over both systems and the resulting visualizations. The authors argue that by combining the information provided by the two tools, queries can be constructed that yield notably more useful information than what is possible if the programs are kept separated.

Furthermore, [ZCW+18] use the term *hybrid provenance* to talk about such combinations of prospective and retrospective provenance. Their work focuses on integrating YesWorkflow's high-level process-view with fine-grained information gained by retrospective provenance recorders. They do not solely concentrate on noWorkflow, but rather on building more general *provenance bridges* in order to connect YesWorfklow with other tools. They demonstrate such a bridge based on DataONE RunManager[1]. This API enables the collection of provenance on file operations and has implementations as a package for R and a MATLAB toolbox.

[1] https://github.com/DataONEorg/sem-prov-design/blob/master/docs/PROV-capture/Run-manager-API.rst

CHAPTER 3

Theoretical Background

The purpose of this chapter is to provide background knowledge essential to this thesis. That is, the terminology and technologies that are crucial for the subsequent chapters will be established concisely.

3.1 Ontologies and Provenance

In the realm of philosophy, *ontology* refers to the study of being, i. e. what exists in a particular area of interest. In Computer Science, however, ontologies formally specify a conceptual model of knowledge in a specific domain. More precisely, they define vocabulary enabling machine-aided classification, interpretation and processing of knowledge, even across multiple information systems. [GAVS11, pp. 509–512]

For instance, an ontology in the field of vehicles offers the capability to express relationships between vehicles and domain-specific properties, as Figure 3.1 illustrates.

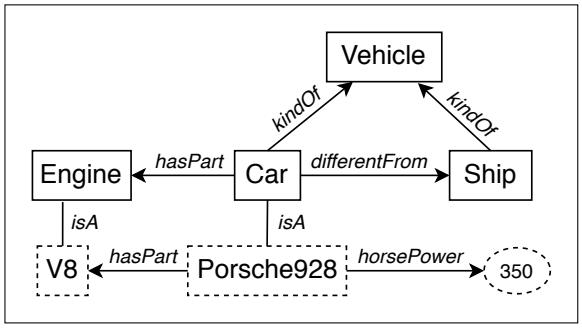

Figure 3.1: Visualization of an exemplary Ontology. From [GAVS11, p. 511]

Both a car and a ship are vehicles, but they are different. Cars have engines and the Porsche928 is an instance of a car that has a V8 engine and horse power 350.

The primary subject of this thesis, provenance of Python scripts, is also nothing other than domain-specific knowledge that may be expressed by means of an ontology. It is well possible to construct different ontologies which essentially describe the same domain (with potentially diverging focal areas). Hence, in regard to the practical part of the work at hand, a provenance ontology had to be decided on. The World Wide Web Consortium's PROV-O was chosen, which will be elaborated on in Section 3.2. Prior to that, however, some terminological notes are necessary.

Up until now, the term *provenance* has been used rather intuitively, without a well-defined meaning. In this thesis, the definition provided by W3C is used. It states that provenance is "*a record that describes the people, institutions, entities, and activities involved in producing, influencing, or delivering a piece of data or a thing*" [MMB+13]. Since this work deals with provenance in the context of scripts and script executions, there must be made a distinction between different types of provenance. As already mentioned, there is *prospective* and *retrospective* provenance – the notion of *hybrid* provenance will be neglected for rest of the thesis.

Prospective provenance may be interpreted as abstract workflow specification. It is typically useful for describing the control and data flow of a script on a high level, i. e. it makes the workflow structure explicit by hiding implementation details. In contrast, retrospective provenance models workflow executions in detail. It records tasks, e. g. functions, that were performed as well as how data artifacts were used and generated by these tasks – in short, it aggregates data derivation information. [LLCF10]

3.2 The W3C PROV Standard

The PROV family of documents, crafted by the W3C Provenance Working Group, consists of eleven documents (plus an overview document) that intend to facilitate the representation and interchange of provenance data on the Web and other information systems. To increase interoperability, they employ widely adopted formats such as XML and RDF, the latter of which will be discussed in Section 3.3. Furthermore, PROV provides means to validating and accessing provenance. [GM13]

Since the aim is to utilize provenance across multiple systems, PROV provides a *conceptual data model* (PROV-DM) that is application- and technology-independent. Heterogeneous systems can export their native provenance in this generic data model, which enables other systems to import this data and process it as desired. [MMB+13] The W3C Provenance Working Group defined three serialization formats for PROV-DM: PROV-N [MMCSR13] aims at high human readability, PROV-XML [ZGH13] defines an XML schema for the PROV data model and PROV-O [LSM+13] introduces a corresponding ontology. PROV-O is an OWL2 ontology, i. e. it allows a mapping from PROV to RDF, which in turn offers multiple serialization formats (see Subsection 3.3.1).

Due to its high relevance for this thesis, the next subsection is dedicated to describing PROV-O in greater detail.

3.2.1 PROV-O

The Provenance Working Group realized that most users' requirements do not warrant the entire ontology. Hence, PROV-O classes and properties (*terms*) come in three groups:

Starting Point Terms: create simple provenance descriptions that might be reified with the help of terms from other groups

Expanded Terms: provide additional information; many of these terms are subclasses or subproperties of those in the Starting Point group

Qualified Terms: specify further information about binary relations from the first two groups, i. e. they *qualify* existing relations

Starting Point Terms are the fundamental constituents of PROV-O. Figure 3.2 illustrates the three classes – entity, activity, agent – and nine properties. Entities are all kinds of physical, digital or conceptual objects. Activities occur over a certain period of time and relate to entities (e. g. by consuming or generating them). Agents are ascribed responsibility for the existence of certain entities or the initiation of an activity. [LSM+13]

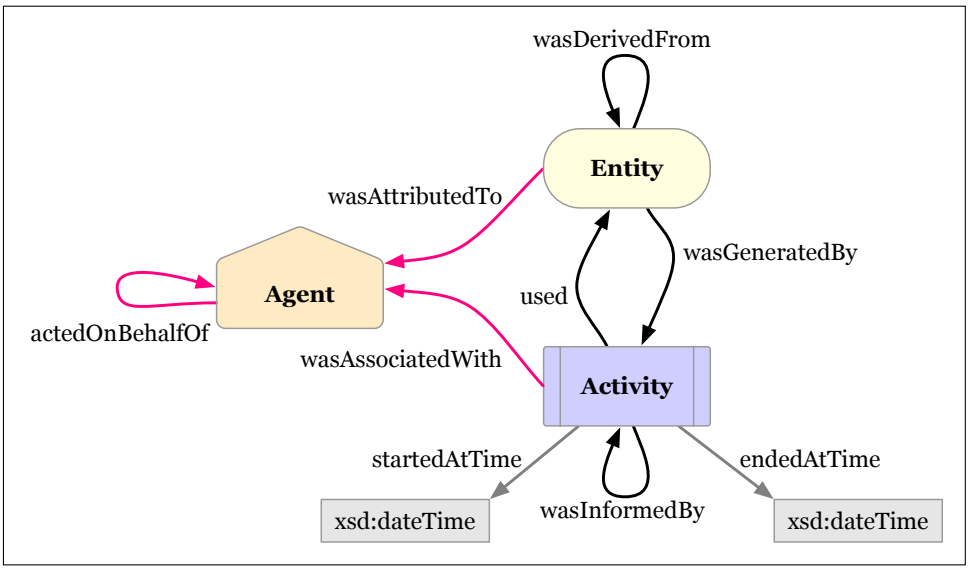

Figure 3.2: PROV-O Starting Point Terms. From [LSM+13]

For a visualization of the Expanded Terms and more precise definitions of members of all three categories, see [LSM+13].

As a side note, since PROV-O has been designed for the World Wide Web, objects are uniquely identified by URIs. For convenience reasons, *qualified names* are used in this context. This notation allows denoting a resource `http://www.example.com/ArtifactA` by `ex:ArtifactA`, where `ex` is the *(namespace) prefix* and `ArtifactA` the *local name*. Keep in mind, however, that prefixes have to be explicitly declared. [MG13, p. 6]

Listing 3.1 demonstrates how to express the PROV-O provenance of a news post containing an article and a graph. It utilizes all three term classes and is formalized in PROV-N. It is a highly simplified and slightly adapted version of the complete example given in [GMB+13]. Moreover, Appendix B presents a visualization of this provenance document.

```
1  document
2   prefix ex <http://example.com/>
3   prefix foaf <http://xmlns.com/foaf/0.1/>
4   prefix dcterms <http://purl.org/dc/terms/>
5
6   agent(ex:derek, [foaf:givenName="Derek", prov:type='prov:Person'])
7   agent(ex:newsproducer, [prov:type='prov:Organization', foaf:name="News Inc."])
8   entity(ex:article, [dcterms:title="Crime rises in cities"])
9   entity(ex:articleV1)
10  entity(ex:articleV2, [prov:label="fixed typo"])
11  entity(ex:dataSet)
12  entity(ex:chart)
13  entity(ex:newsPost)
14  specializationOf(ex:articleV1, ex:article)
15  specializationOf(ex:articleV2, ex:article)
16  wasDerivedFrom(ex:articleV2, ex:articleV1, -,-,-, [prov:type='prov:Revision'])
17  activity(ex:composePost, -, -)
18  activity(ex:collectData, 2012-03-31T09:21:00+01:00, 2012-03-31T12:15:00+01:00)
19  wasAssociatedWith(ex:composePost, ex:derek, -)
20  actedOnBehalfOf(ex:derek, ex:newsproducer, ex:composePost)
21  used(ex:composePost, ex:dataSet, -, [prov:role='ex:dataToCompose'])
22  wasGeneratedBy(ex:newsPost, ex:composePost, 2012-03-31T15:30:00+00:00)
23  wasGeneratedBy(ex:dataSet, ex:collectData, -)
24  wasDerivedFrom(ex:chart, ex:dataSet, -, -, -)
25  wasDerivedFrom(ex:articleV1, ex:dataSet, -, -, -)
26  wasAttributedTo(ex:chart, ex:derek)
27 endDocument
```

Listing 3.1: PROV-O Example in PROV-N Syntax. Adapted from [GMB+13]

Lines 2–4 declare the prefixes used later on. This shows that combining PROV with other vocabularies – *Friend of A Friend* foaf[1] and *Dublin Core* dcterms[2] – is straightforward. PROV-N does not require the `prov` prefix to be declared explicitly.

Derek, a person, has composed a news post for his company News Inc. (lines 6, 7, 17, 19, 20, 22). On that account, several artifacts have been created (lines 8–13). He had to

[1] http://xmlns.com/foaf/spec/
[2] https://www.dublincore.org/specifications/dublin-core/dcmi-terms/

issue two versions of the news article – both are instances of a conceptual article entity that is part of the post – because he did not proofread rigorously enough (lines 14–16).

Besides the article, the post also contains a chart that was produced based on a data set Derek had collected over the course of about three hours (lines 18, 21, 23, 24). Of course, the article referred to these pieces of data, too (line 25). Ultimately, it is explicitly stated that the generation of the chart was conducted by Derek (line 26).

As mentioned above, PROV-O can be mapped to RDF. This is of substantial importance for the work at hand. Thus, the next section elaborates on RDF and how it will be utilized in the remaining chapters.

3.3 RDF and SPARQL

3.3.1 RDF Data Model and Serialization Formats

The *Resource Description Framework* (RDF) specifies a data model for describing resources on the Web. It enables the representation of information that is processed across information systems rather than only displayed to human users. Owing to RDF's abstract model, such data may be exchanged between and made available to numerous applications, even ones the data was not initially designed for. [SR14]

An RDF statement is a triple of the form `(subject, predicate, object)`. Subject and predicate are IRIs, a generalization of URIs to the Unicode character set. Objects can be IRIs or literals, the latter of which may be annotated with a data type as well. A set of RDF *triples* is called an RDF *graph*. In such a graph, subjects and objects are represented by vertices that are connected by directed edges, the predicates. [CWL14]

Table 3.1 shows six triples and Figure 3.3 the corresponding graph. Resources are once again denoted by qualified names; the namespaces behind the prefixes are defined in Listing 3.2, which will be discussed later on.

Subject	Predicate	Object
exr:doc	rdf:type	exs:Report
exr:doc	dc:creator	exr:Fabien
exr:Fabien	rdf:type	foaf:Person
exr:Fabien	foaf:name	"Fabien Muller"^^xsd:String
exr:doc	exs:nbPages	"23"^^xsd:integer
exr:doc	exs:theme	exr:SemanticWeb

Table 3.1: Exemplary RDF Triples. Adapted from [GKHT11, pp. 124–125]

One can observe that, at least for not too large amounts of information, gaining an understanding of the structure of the data is easier using the graphical representation. In Figure 3.3, subjects and IRI objects are illustrated as ovals, with their abbreviated IRIs as labels, e. g. `exr:Fabien`. The same applies to predicate IRIs such as `foaf:name`. The

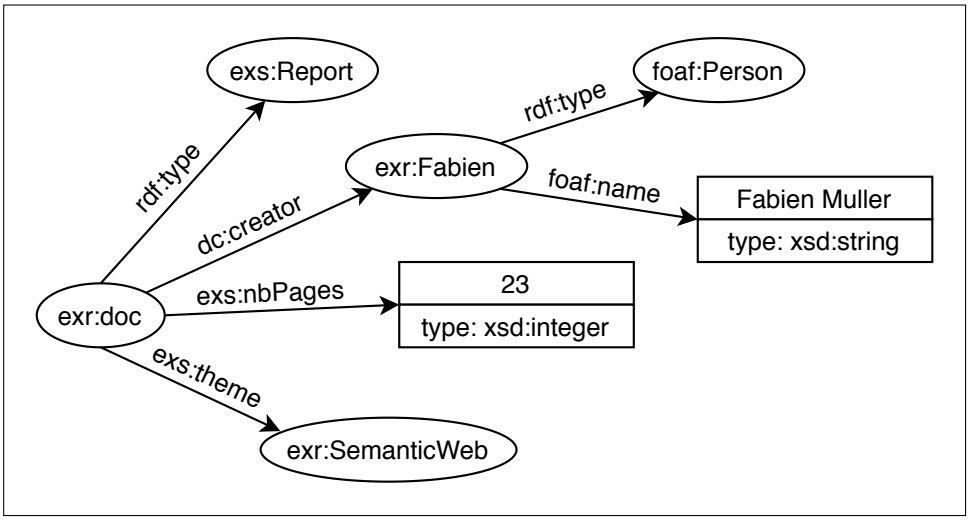

Figure 3.3: RDF Graph based on Table 3.1. Adapted from [GKHT11, pp. 124–125]

predicates are represented by arrows and are responsible for the graph's web-like structure by linking information together. Lastly, literals are shown as rectangles featuring their values and data types, e. g. `Fabien Muller` is an `xsd:string`.

RDF is crucial to this thesis since data in one of the specified serialization formats can be effectively leveraged thanks to the abundant tooling support (see also Chapter 6). The reader is invited to discover all serialization formats available – one might refer to [SR14]. However, Turtle is of particular interest to the work at hand, owing to its syntactic similarities to SPARQL (see Subsection 3.3.2). Hence, Listing 3.2 depicts the six triples from Table 3.1 in Turtle syntax. This notation expresses `(s, p, o)` triples as `s␣p␣o`, succeeded by a period character. It offers syntactic shortcuts such as compactly stating consecutive triples with equal subjects by using semicolons instead of periods. Moreover, the `@prefix` directive enables one to use qualified names to refer to resources.

```
1  @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
2  @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
3  @prefix exr: <http://example.org#> .
4  @prefix exs: <http://example.org/schema#> .
5  @prefix foaf: <http://xmlns.com/foaf/0.1/> .
6  @prefix dc: <http://purl.org/dc/elements/1.1/> .
7
8  exr:doc rdf:type exs:Report ; dc:creator exr:Fabien .
9  exr:Fabien rdf:type foaf:Person ; foaf:name "Fabien Muller"^^xsd:string .
10 exr:doc exs:nbPages "23"^^xsd:integer ; exs:theme exr:SemanticWeb .
```

Listing 3.2: Turtle Representation of Table 3.1. Adapted from [GKHT11, p. 129]

3.3.2 Querying RDF using SPARQL

This introduction to the *SPARQL Protocol and RDF Query Language* (SPARQL) is kept intentionally short because the language has a very descriptive syntax with resemblance to SQL and intuitive semantics. The RDF data from Listing 3.2 is used as sample data. Elaborations on language specifics of SPARQL in this subsection are taken from [HS13].

With SPARQL, one specifies (mostly) conjunctive *triple patterns* that have to be matched by result entries. A triple pattern is essentially an ordinary triple, but subject, predicate and object may be variables, denoted by a leading question mark (e. g. `?x rdf:type ?y .`). A query result is a sequence of variable mappings that satisfy all patterns.

Listing 3.3 depicts a very basic SPARQL query. It selects all objects with subject `?x` that are of type `?y` *and* are associated with a name `?z`. In the sample data set, there are only two subjects, `exr:doc` and `exr:Fabien`. Since the former does not exhibit a name, the result set is constituted of only one variable mapping, as shown by Table 3.2.

```
1 PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3
4 SELECT (?x AS ?object) (?y AS ?type) (?z AS ?name) WHERE {
5  ?x rdf:type ?y .
6  ?x foaf:name ?z .
7 }
```

Listing 3.3: Simple, purely conjunctive SPARQL Query

object	type	name
exr:Fabien	foaf:Person	"Fabien Muller"

Table 3.2: Result of SPARQL Query from Listing 3.3

In order to conclude this section, the query in Listing 3.4 employs several useful language features. Its aim is to find all properties of the two objects from the sample data set.

Lines 5 and 6 select all objects with a defined type and probably different, possibly multiple properties. Line 7 expresses a disjunction: The bracketed pattern does not need to be matched, i. e. exhibiting a name is optional. Subsequently, line 10 states that the first of the terms `?n` and `"<none>"` which can be evaluated without error ("non-null") is bound as new variable `?name`. Since both objects from the sample data set satisfy line 5 and line 6 is trivially always satisfied, some triples must be filtered out so that the variable `?property` provides useful information. This is ensured by line 9 which removes triples with the subject `rdf:type` from the result set.

In order to keep the result compact (see Table 3.3), it is grouped by the objects and their corresponding name, a mechanism commonly known from SQL (line 12). The

GROUP_CONCAT call in line 4 concatenates the aggregated properties to a string with a specifiable separator character which defaults to a space.

```
1  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2  PREFIX foaf: <http://xmlns.com/foaf/0.1/>
3
4  SELECT ?object ?name (GROUP_CONCAT(?property) AS ?properties) WHERE {
5   ?object rdf:type ?y .
6   ?object ?property ?z .
7   OPTIONAL { ?x foaf:name ?n . }
8
9   FILTER(?property != rdf:type)
10  BIND(COALESCE(?n, "<none>") AS ?name)
11 }
12 GROUP BY ?object ?name
```

Listing 3.4: SPARQL Query with disjunctive Constructs

The result shows that exr:doc is not associated with a name, but has three properties besides rdf:type and exr:Fabien does exhibit the name "Fabien Muller", but has only one additional property.

object	name	properties
exr:doc	"<none>"	exs:nbPages exs:theme dc:creator
exr:Fabien	"Fabien Muller"	foaf:name

Table 3.3: Result of SPARQL Query from Listing 3.4

There are many more features to SPARQL that have not been discussed, some of which will be made use of in Chapter 6. Should their meaning not be evident from the given context, there will either be given brief explanations or relevant literature will be referenced.

3.4 Summary

This chapter served as an introduction to the concept of provenance and its ontological representation using PROV-O. Additionally, the foundations of the Resource Description Framework (RDF) along with SPARQL as its query mechanism were explained. These notions and technologies are crucial for the subsequent chapters as they mostly revolve around expressing YesWorkflow and noWorkflow output in PROV-O and examining how to make use of this data.

CHAPTER 4

Solution Concept

The following sections depict the strategy to achieve the goals set for this thesis. This includes the general concept, development efforts, considerations behind the implementation as well as technologies and tools that were employed.

4.1 Sample Scripts

Since the work at hand deals with provenance capturing in the context of Python scripts, there is a requirement for exemplary scripts which may be used as input for YesWorkflow and noWorkflow. Three such software artifacts have been conceived, each of which exhibits different properties that are interesting for testing and demonstration purposes. Furthermore, they are increasing in size and complexity (either computationally or logically). For better readability and in order not to waste space, their source code is referenced via GitHub.

4.1.1 Small-Sized Instance

The smallest test script is an example of very basic data processing with few dependencies and not too sophisticated logic. Its source code, along with exemplary input and output data, can be found on GitHub[1].

The script takes wind speed measurements from cities in various countries as input and classifies them according to the Beaufort scale [The17], a thirteen-level empirical measure for wind forces. Afterwards, the script's output is constituted by writing two files to the local storage. While the first one lists the ten countries with the highest average wind speeds, the latter one consists of the the ten highest wind speed measurements overall.

[1] https://github.com/raffaelfoidl/ProvCaptPyEnvs/tree/master/instances/small

The script's usage is illustrated by Listing 4.1.

```
usage: inst_s.py data

positional arguments:
  data    input file path (CSV with country, code, city,
          wind_speed)
```

Listing 4.1: Small Sample Script: Usage Message and Argument Descriptions

It expects a CSV file with exactly four columns as input: country name, two-letter country code (e. g. `AT` for Austria), city name and measured wind speed. The output files are JSON arrays, their file paths are predefined in the script's source code.

4.1.2 Medium-Sized Instance

In terms of lines of code, this test script is smaller than the one presented in the previous subsection. However, it features more – especially transitive – module dependencies and has a higher computational complexity. This may make the process of capturing provenance more expensive. Hence, this instance is still regarded as the medium-sized one. Its source code, exemplary input and output files are again available on GitHub[2].

The script applies a Gaussian blur filter to an arbitrary image and compares the result to the original. It calculates the absolute value of the color difference between corresponding pixels and, based on that, creates a third image – the difference image. As an example, if a white pixel in the original image has become a black one (or vice versa) in the blurred image, the difference image is white at this particular pixel. If, by contrast, there is no change in color detected, the corresponding pixel in the difference image is black.

Listing 4.2 depicts the script's usage.

```
Usage: inst_m.py sigma in out diff

positional arguments:
  sigma    sigma for the Gaussian blur
  in       input file path
  out      output file path
  diff     diff file path
```

Listing 4.2: Medium Sample Script: Usage Message and Argument Descriptions

An invocation requires an integer-valued standard deviation `sigma` (symbolically σ) for the Gaussian kernel calculation, the input image's file path and file paths the output images should be written to. All common image file formats are supported.

[2] https://github.com/raffaelfoidl/ProvCaptPyEnvs/tree/master/instances/medium

4.1.3 Large-Sized Instance

The last test instance is modularized into multiple files due to its size and many processing steps. Moreover, it takes advantage of external libraries, model classes and reads from and writes to multiple files. Thus, it exhibits a relatively high complexity compared to the small and medium instance. Once more, source, input as well as output files can be found on GitHub[3].

The script reads a number of process *traces* which hold information about a process execution (process ID, start time, end time, accessed resources, read or write mode). They are used to identify possible sources of resource conflicts for a specific process. In this context, a *possible resource conflict* is defined as two processes with overlapping execution times having write access to the same resource(s) at any, unspecified point in time, i. e. actual concurrency cannot be inferred from the traces alone. This test instance's output depends on the process ID specified and is comprised of one plot depicting the number of traces per process and one illustrating the specified process's accesses per resource. Additionally, the detected possible sources of resource conflicts are written to a file, along with the data underlying the resource usage plot.

In Listing 4.3, the script's usage can be observed.

```
Usage: inst_l.py pid freqs summary_csv summary_plot conflicts
                 traces [traces ...]

positional arguments:
  pid             ID of the process to analyze
  freqs           path frequency plot should be persisted to
  summary_csv     path resource summary (csv) should be
                  persisted to
  summary_plot    path resource summary (plot) should be
                  persisted to
  conflicts       path conflict report should be persisted to
  traces          process trace logs (JSON) to analyze
```

Listing 4.3: Large Sample Script: Usage Message and Argument Descriptions

When calling the script, the ID of the process for which possible resource conflicts should be discovered is required. Furthermore, desired output file paths are expected as well as at least one file with process traces as input.

[3] https://github.com/raffaelfoidl/ProvCaptPyEnvs/tree/master/instances/large

4.2 YesWorkflow

4.2.1 Proposed Modifications

YesWorkflow provides mechanisms to export the workflows defined by the annotations as either Prolog facts or dot files. The latter option was chosen to render the Figure from Appendix A using Graphviz dot[4].

This thesis proposes a new extractor module for YesWorkflow that composes provenance information adhering to the W3C PROV Ontology. Subsequently, it provides the program with new output formats such as PROV-N, Turtle, XML, RDF/XML.

Each annotation that YesWorkflow currently offers is transformed into a corresponding PROV-O counterpart. As a consequence, this new provenance extractor module by definition offers at least the same amount of information as the already existing graph module. In fact, it uses the data retrieved by the annotation parsing module more extensively, allowing for a more thorough overview than what was possible before. Subsection 5.1 goes into more detail about this.

4.2.2 Course of action

There are two YesWorkflow repositories available on GitHub, yw-prototypes[5] and yw[6]. While the first one is written in Java and was developed as a prototype, the latter one is a C++ implementation resulting in native executables. At the time of writing this thesis, the development of the native edition appears to be stagnant, and hence does not offer advantages over the prototype. Furthermore, there is better PROV-O library support available for Java. As a consequence, the prototype was chosen as starting point for the modifications proposed.

Initially, the repository was forked from GitHub. In order to maintain traceability, all development work was done on a separate branch provenance-export, which was created from the latest commit on master. After completing the implementation, provenance-export was merged back into master.

As mentioned above, a library for conveniently assembling PROV-O compliant output was utilized. ProvToolbox[7] supports reading, writing and converting provenance data of various formats that are defined by the PROV standard. It is deployed via the Maven Central Repository; the configuration for using it in a project is depicted in Listing 4.4.

[4] https://www.graphviz.org/
[5] https://github.com/yesworkflow-org/yw-prototypes
[6] https://github.com/yesworkflow-org/yw
[7] https://lucmoreau.github.io/ProvToolbox/

```
<dependency>
  <groupId>org.openprovenance.prov</groupId>
  <artifactId>prov-model</artifactId>
  <version>0.9.2</version>
</dependency>
<dependency>
  <groupId>org.openprovenance.prov</groupId>
  <artifactId>prov-interop</artifactId>
  <version>0.9.2</version>
</dependency>
```

Listing 4.4: Maven Configuration for ProvToolbox

The prov-model artifact allows developers to make use of the PROV data model using Java; prov-interop enables converting between the PROV data model and serializations specified by the standard (and vice versa).

The source code containing all modifications and extensions that have been made to the original yw-prototypes repository as part of this thesis may be found on GitHub[8].

4.3 noWorkflow

4.3.1 Proposed Modifications

noWorkflow's export functionality is limited in the sense that the supported output mediums to describe the three types of provenance collected during a trial execution are Prolog facts, a textual representation and Jupyter Notebooks.

The modifications proposed in this thesis extend noWorkflow such that output formats which adhere to the W3C PROV Ontology are also available. As with YesWorkflow, PROV-N, Turtle, XML, RDF/XML and more is supported. Not only can the collected provenance of a single trial be exported in a PROV-O compliant way, but also the comparison of two trial executions.

To this end, a new command `now provo` has been introduced, along with options resembling the ones available for `now show`. Furthermore, a new option switch has been added to `now diff` that generates an output comparison file in the PROV-O format.

4.3.2 Course of Action

As with YesWorkflow, at first, noWorkflow's repository was forked from GitHub[9]. Similarly, the implementation was carried out on a new, separate prov-o branch which was merged back to master after completion.

[8] https://github.com/raffaelfoidl/yw-prototypes
[9] https://github.com/gems-uff/noworkflow

However, in noWorkflow's official repository, the master branch is not the most current one, but 2.0-alpha. Despite latest development efforts with several improvements (e. g. support for Python 3.6 and higher) having been pushed only to 2.0-alpha, the implementation of the extensions proposed in this thesis was started off from the master branch. This decision is mainly founded upon two facts. Firstly, in the local environment, certain stability problems were encountered with 2.0-alpha. Secondly, there already have been made efforts for a standardized provenance output. For this feature, though, an extended variant of PROV-O – Versioned-PROV[10] – is employed. With the intention of not discarding already existing functionality nor creating ambiguity with two different provenance export modules, the master branch was chosen.

noWorkflow is written in Python which, like Java, provides library support for W3C PROV-O. More specifically, the prov package[11] provides means for taking advantage of the PROV data model, reading and creating provenance documents as well as exporting them to files in different serializations. It is available at PyPI and may be installed using pip, as illustrated by Listing 4.5.

```
pip install prov
```

Listing 4.5: Installing the prov package into a (virtual) Python environment

Again, the source code with all modifications and extensions made to noWorkflow as part of this thesis is available on GitHub[12].

4.4 SPARQL Playground

Providing an implementation for generating output for YesWorkflow and noWorkflow that conforms to the W3C PROV standard addresses the first research question of this thesis. After having achieved this goal, one might want to find ways of effectively utilizing the provenance data generated, which corresponds to the second research question. For this purpose, the RDF serialization format Turtle will be taken advantage of, in combination with SPARQL.

As will be elaborated in Chapter 6, there are common questions and use cases one might encounter when analyzing workflow or, in this context, script executions. There will be referenced, and in some cases directly presented, SPARQL queries that give answers to such problems. In conjunction with these queries, an adapted version of the SPARQL Playground[13], which is being developed at the Swiss Institute of Bioinformatics, is available. It provides the reader with an easy way to experiment with the queries using their own data and/or execute modified queries. Figure 4.1 illustrates the customized variant of the tool.

[10] https://dew-uff.github.io/versioned-prov/index.html
[11] https://github.com/trungdong/prov
[12] https://github.com/raffaelfoidl/noworkflow
[13] http://sparql-playground.sib.swiss/

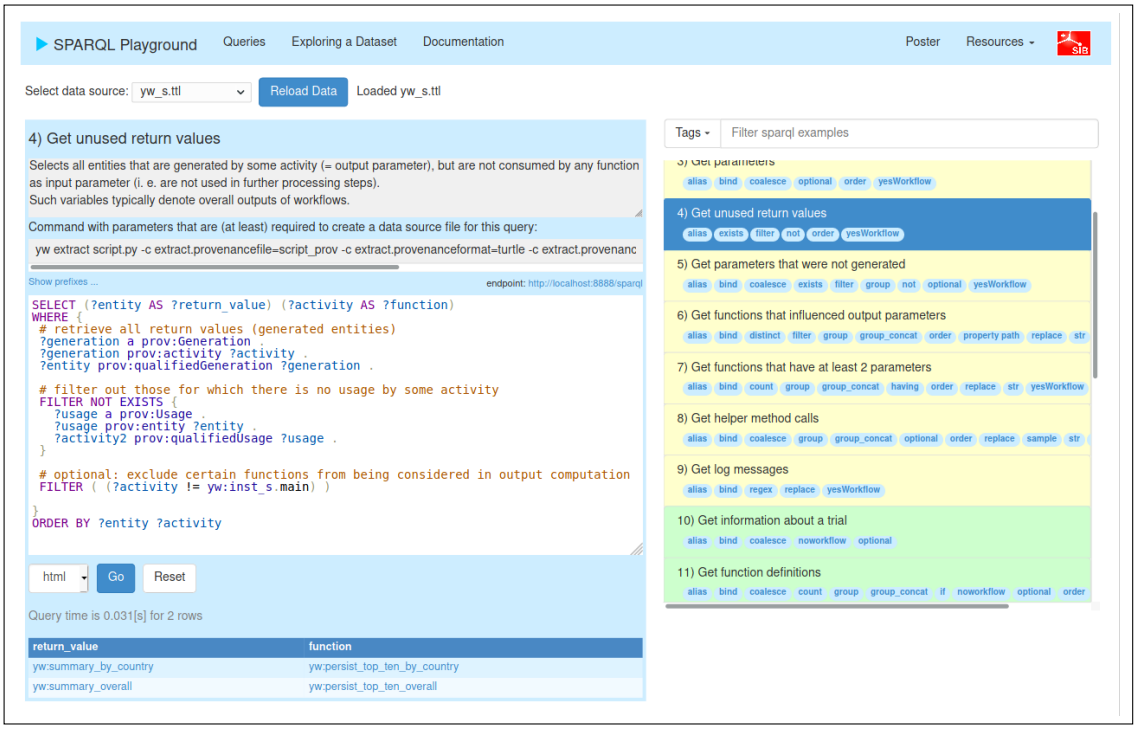

Figure 4.1: SPARQL Playground for querying collected Provenance Data

It es equipped with all the queries that have been formulated in the course of the work at hand. Their titles and associated tags (e. g. relevance for YesWorkflow, noWorkflow or language features used) can be observed on the right hand-side of the page. After selecting a query, the left-hand side indicates a description, a command for generating suitable test data and the SPARQL query itself. The Turtle input file is selected and loaded at the top of the website. Finally, the queries are executed using the respective button at the bottom left of the page; their results are displayed in a table underneath.

The SPARQL Playground is a Java Spring Boot application with an AngularJS frontend and uses Sesame[14] as query engine. Listing 4.6 demonstrates how it can be started.

```
java -Dserver.port=8888 -jar sparql-playground.war
```

Listing 4.6: Command for starting the SPARQL Playground

Once the startup has completed, the application may be utilized using the browser at http://localhost:8888. Its source code, the queries, test data files and notes regarding how it has been altered compared to the original version can be found on GitHub[15].

[14] https://bitbucket.org/openrdf/sesame/src/master/
[15] https://github.com/raffaelfoidl/sparql-playground

4.5 Summary

This chapter gave an overview of how the research questions of this thesis are going to be answered. To this end, the extensions to be implemented for YesWorkflow and noWorkflow have been defined (see Chapter 5). Furthermore, the three exemplary Python scripts that are used to test and demonstrate the new features were described. Ultimately, the SPARQL Playground was introduced which is employed to query PROV-O files generated by the two tools (see Chapter 6).

CHAPTER 5

Implementation Description

This chapter explains how the proposed modifications to YesWorkflow and noWorkflow were achieved. It gives a sufficiently detailed overview of the design decisions and key strategies of the implementation. Recalling the goals of the work at hand, this chapter answers the first research question.

5.1 YesWorkflow

The work to be done in order to establish PROV-O compliance for YesWorkflow is predetermined by the annotations available. For each of them, there has to be defined a mapping to corresponding PROV-O terms. This section explains general modifications made to the application, the overall approach to extracting provenance information and shows how each annotation is modeled in PROV-O.

At first, new command line options were introduced in order to make the new provenance extractor module accessible. They are depicted by Table 5.1.

Configuration Name	Description
`extract.provenancefile`	File for storing provenance information about scripts (no extension)
`extract.provenanceformat`	Format of provenance information: provn, turtle, xml, rdfxml, trig, json, pdf, svg, dot, png, jpeg
`extract.provenancens`	Namespace for provenance information about scripts
`extract.provenanceprefix`	Namespace prefix for provenance information about scripts

Table 5.1: New Command Line Options for YesWorkflow

After setting up the ProvToolbox library to make use of the PROV data model, as already shown by Listing 4.4, a strategy for extracting provenance was conceived. For this purpose, new model classes – `AnnotationBlock` and `AnnotationLine` – have been introduced. An `AnnotationBlock` instance represents one block of annotated source code comments and consists of multiple `AnnotationLine` instances. They are populated with Annotation objects that have been created by YesWorkflow's parser module.

Once the list of `AnnotationBlock` objects has been instantiated, the provenance document can be built based on it. Algorithm 5.1 demonstrates roughly how these pieces of data are converted to PROV-O in pseudocode.

Algorithm 5.1: Provenance Document Generation for YesWorkflow

Input: List of `AnnotationBlock` objects `blocks`
Output: PROV-O Document `doc` with at least as much information as `blocks`

```
1  ProvDocument doc = new ProvDocument();
2  foreach AnnotationBlock block ∈ blocks do
3      foreach AnnotationLine line ∈ block.getLines() do
4          foreach Annotation annotation ∈ line.getAnnotations() do
5              ProvInfo prov = annotation.getProvenanceInfo();
6              ProvInfo extraProv = typeSpecificInfo(doc, prov, annotation.type());
7              doc.add(prov);
8              doc.add(extraProv);
9          end
10     end
11 end
12 return doc;
```

For each code block, every line and every annotation must be handled in order to process all annotations that are defined in a script (lines 2–4). Line 5 extracts the PROV-O representation of the current annotation (e. g. `entity` for `@in`). Line 6 creates additional provenance records that are specific to the type of the current annotation and might also include existing records (e. g. connecting functions with input arguments via `used`). More information on these two lines is given below. Subsequently, the constructed provenance records are added to the document (lines 7, 8). After the last iteration, the document is returned (line 12) and may be serialized to a file using ProvToolbox.

One short note on runtime: Realistically, only the outer loop must be expected to have a potentially large number of iterations (number of code blocks). The middle loop (number of lines per block) is bounded approximately by the number of YesWorkflow annotations – for blocks with many parameters, the actual number might be slightly higher. Lastly, the inner loop (number of annotations per line) should not exhibit many iterations due to otherwise ensuing readability drawbacks in the respective script. Thus, this naive algorithm performs justifiably decent. However, efficiency could still be increased with more elaborate approaches.

Table 5.2 illustrates how each of YesWorkflow's annotation has been modeled in PROV-O. For instance, occurrences of `@begin` annotations are transformed into PROV-O activities. They may be connected with entities such as function calls or input parameters, both of which are modeled as entities. Likewise, output parameters defined by `@out` or `@return` are linked with the generating activities via `wasGeneratedBy` records. Qualifications such as `@desc` are represented by labels, adding clarifying information to the entities or activities that arise from their primary annotations. An exception to this is `@as` which results in an `alternateOf` record.

Annotation	Type	PROV-O term	Extra (other Record)
`@begin`	standalone	`activity`	`used` (`entity`)
`@end`	standalone	nothing	
`@as`	qualification	`alternateOf`	
`@call`	standalone	`entity`	
`@desc`	qualification	`label`	
`@file`, `@uri`	qualification	`label`	
`@in`, `@param`	standalone	`entity`	
`@out`, `@return`	standalone	`entity`	`wasGeneratedBy` (`activity`)
`@log`	qualification	`label`	

Table 5.2: YesWorkflow Annotation Mappings to PROV-O Terms

A visualization of the provenance document generated from the small example script, along with more output files produced by the extended version of YesWorkflow, can be retrieved from GitHub[1].

As has already been hinted at in Subsection 4.2.1, the new provenance extractor module utilizes the data collected by the parser module more thoroughly than the graph module. For example – unlike the graph module – when employing the `@as` annotation, the provenance extractor does denote the element being aliased. Similarly, the graph module does not visualize `@call`, `@log` nor `@desc` annotations. In contrast, the provenance extractor module does express them in its PROV-O output – even for aliased entities. Lastly, if elements occur in more than one code block, each qualification (`@desc`, `@file`, `@uri`) adds a new label, i. e. they do not override each other. This may be useful for providing context-sensitive information. For example, the argument `data_by_country` from the small test instance is used both as input and as output parameter with different descriptions.

The modified source code of YesWorkflow with all extensions that have been implemented for this thesis can be found on GitHub[2].

[1] https://github.com/raffaelfoidl/ProvCaptPyEnvs/tree/master/yesworkflow_outputs

[2] https://github.com/raffaelfoidl/yw-prototypes

5.2 noWorkflow

For noWorkflow, a new command `now provo` has been introduced. It has essentially the same functionality as `now show`, but instead of printing the provenance data to the standard output, it is exported to a file in a PROV-O compliant way using the prov package. Additionally, the `now diff` command has received a new command line option that triggers the trial comparison to be generated as PROV-O document. Table 5.3 summarizes the most important command line options that have been added to noWorkflow in the course of implementing these features.

Command	Option	Description
`provo`	`-m`	export module dependencies
`provo`	`-d`	export user-defined functions, global variables and function calls
`provo`	`-e`	export environment conditions
`provo`	`-f`	export read/write accesses to files
`provo`	`-a`	export function activations
`provo`	`-r`	set maximum recursion depth for analysis of function activations within function activations
`diff`	`-p`	export comparison as PROV-O document
`provo, diff`	`-n`	set default namespace for exported PROV-O file
`provo, diff`	`--file`	set name/path of PROV-O file to be exported (no extension)
`provo, diff`	`--format`	set format of exported PROV-O file; allowed values provn, turtle, xml, rdfxml, trig, json, pdf, svg, dot, png, jpeg

Table 5.3: Most important new Command Line Options for noWorkflow

For example, in order to export the environment conditions of trial 3 to an XML file, noWorkflow should be invoked using `now provo -e --format xml 3`.

In contrast to YesWorkflow, for noWorkflow, there was no need to create new data structures like `AnnotationBlock` or `AnnotationLine`. All information the program records during a trial execution (recall Figure 2.1) was already represented in such a way that it could be immediately reused for the purposes of this thesis.

However, prior to the actual implementation, a bug in the retrieval of definition provenance had to be resolved. For function parameters with type hints (i. e. according to PEP-484[3]), noWorkflow erroneously recognized the type annotations instead of the arguments. For example, analyzing a function with the signature `foo(bar: Sequence[Tuple[str, int]])` yielded four arguments `Sequence`, `Tuple`, `str`, `int` instead of only one parameter `bar`.

As mentioned above, noWorkflow already stored the provenance information in suitable data structures. This fact allowed for a uniform approach of creating PROV-O records

[3] https://www.python.org/dev/peps/pep-0484/

across all types of information. Algorithm 5.2 illustrates the idea behind this approach using the example of selected parts of the definition provenance.

Algorithm 5.2: Provenance Record Generation for noWorkflow

Data: Recorded Provenance of a Script Execution `trial`
Result: PROV-O Document `document` contains Function Fefinitions of `trial`

```
1  foreach FunctionDef func ∈ trial.function_defs do
2      document.newActivity("functionDefinition", func.id, [... attributes ...]);
3      foreach ArgumentDef arg ∈ func.arguments do
4          document.entity("argumentDefinition", arg.id, [... attributes ...]);
5          document.used("argumentDefinition", func.id, arg.id, [... attributes ...]);
6      end
7      foreach GlobalDef glob ∈ func.globals do
8          document.entity("globalDefinition", glob.id, [... attributes ...]);
9          document.used("globalDefinition", func.id, glob.id, [... attributes ...]);
10     end
11 end
```

Every function that was registered during the abstract syntax tree analysis is mapped to an activity (line 2). Arguments and global variables are represented as entities (lines 4, 8). The fact that they are used by the functions is expressed accordingly (lines 5, 9).

For the other types of provenance, only minimal adjustments to this approach have to be made. The only exception to this are function activations (execution provenance) because they are triggered by nested function calls. The algorithmic idea to create provenance records for them is very similar to the one depicted in Algorithm 5.2. The primary difference is a recursive formulation rather than an iterative one.

The algorithm already outlines how noWorkflow's prospective provenance is modeled in PROV-O. Table 5.4 gives an overview of mappings from the most important pieces of noWorkflow data to their corresponding PROV-O representations.

For example, environment attributes (command line option `-e`) such as the operating system or hardware specifications are stored in a `collection`. Its members are `entity` records indicating attribute names and values. They are linked together via `hadMember` instances. Similarly, return values of function activations (command line option `-a`) are `entity` records and linked to their respective functions using `wasGeneratedBy` statements.

All information stored in noWorkflow's already existing output still remains in the PROV-O export. Additionally, concerning function definitions, documentation strings and the location in the source code (first and last line of functions) are now expressed, too. Furthermore, the `provo -r` command line option – as shown in Table 5.3 – sets a threshold that determines whether function activations performed by a specific function should be exported (maximum recursion depth). This might be helpful if there are many function invocations whose provenance is not relevant, e. g. in a nested loop.

Type	PROV-O Terms	Attributes: Values
Environment Condition	`entity`, `collection`, `hadMember`(*collection, environmentAttribtue*)	`label`: attribute name, `value`: attribute value, `type`: *environmentAttribute*
Module Dependency	`entity`, `collection`, `hadMember`(*collection, moduleDependency*)	`label`: module name, `location`: module path, `type`: *moduleDependency*
Function Definition	`activity`	`label`: function name, `type`: *functionDefinition*
Argument Definition	`entity`, `used`(*functionDefinition, argumentDefinition*)	`label`: argument name, `role`: argument, `type`: *argumentDefinition*
Global Variable	`entity`, `used`(*functionDefinition, globalDefinition*)	`label`: variable name, `type`: *globalDefinition*
Function Call within Function	`activity`, `wasInformedBy`(*callDefinition, functionDefinition*)	`label`: called function, `type`: *callDefinition*
Function Activation	`activity`, `wasInformedBy`(*functionActivation, functionActivation*)	`starttime`: function start, `endtime`: function finish, `label`: function name, `type`: *functionActivation*
Argument Activation	`entity`, `used`(*functionActivation, argumentActivation*)	`label`: argument name, `value`: argument value, `type`: *argumentActivation*
Global Variable Activation	`entity`, `used`(*functionActivation, globalActivation*)	`label`: variable name, `value`: variable value, `type`: *globalActivation*
Return Value	`entity`, `wasGeneratedBy`(*returnValue, functionActivation*)	`value`: return value, `type`: *returnValue*
Replaced Attribute *(diff)*	`entity`, `wasGeneratedBy`(*attr2, trial2*), `wasInvalidatedBy`(*attr1, trial2*), `wasRevisionOf`(*attr2, attr1*)	`label`: attribute name, `value`: attribute values

Table 5.4: PROV-O Modeling of major noWorkflow Provenance Components

Exemplary outputs for the test scripts generated by the new noWorkflow provenance exporter module are available on GitHub[4], as well as the extended source code[5].

[4] https://github.com/raffaelfoidl/ProvCaptPyEnvs/tree/master/noworkflow_outputs

[5] https://github.com/raffaelfoidl/noworkflow

5.3 Summary

In this chapter, it was outlined how the extensions proposed for YesWorkflow and noWorkflow were implemented. More specifically, in addition to the algorithmic ideas behind the PROV-O export, it was stated how the different components of the programs' outputs have been modeled in this ontology. The following chapter demonstrates the validity of this representation as well as possible possible ways of utilizing it.

CHAPTER 6

Utility Assessment

In this chapter, justifications are given as to why the output generated by the extended versions of YesWorkflow and noWorkflow conforms to the PROV standard. Furthermore, it will be explored in which ways the provenance collected by these tools can now be utilized more effectively. It will also be shown how information gathered by YesWorkflow and noWorkflow might differ. This answers the second research question of this thesis.

6.1 Standard Compliance

The newly implemented extensions to the two programs selected for this thesis are only useful if the produced output is compliant to PROV-DM.

PROV-CONSTRAINTS [CMMDN13] specifies rules and restrictions which provenance data must satisfy in order to be considered valid. They are classified in four groups: *uniqueness*, *event ordering*, *impossibility* and *type* constraints. For example, Constraint 37[1] is an event ordering constraint and states that if an entity is `used` by some activity, there must be a preceding `wasGeneratedBy` event for that entity. Likewise, Constraint 51[2] expresses an impossibility by postulating that entities must never be specializations of themselves.

Furthermore, the attributes that are predefined within the PROV namespace are not arbitrarily applicable. Table 6.1 demonstrates these restrictions.

In order to prove that the produced YesWorkflow and noWorkflow outputs are valid, one could – in theory – manually verify that they satisfy the constraints defined for PROV. However, this not only very tedious, but also practically infeasible. To this end, the PROV Implementation Report [ZGH13] provides helpful information. Besides an extensive list of programs implementing the PROV standard, this document also mentions three

[1] https://www.w3.org/TR/prov-constraints/#generation-precedes-usage
[2] https://www.w3.org/TR/prov-constraints/#impossible-specialization-reflexive

Attribute	Allowed In
`label`	any construct
`location`	Entity, Activity, Agent, Usage, Generation, Invalidation, Start, End
`role`	Usage, Generation, Invalidation, Association, Start, End
`type`	any construct
`value`	Entity

Table 6.1: Domains of PROV Attributes. Adapted From [MMB+13]

tools that may be used to validate provenance data: prov-check (supports PROV-O)[3], Prov-Validator (supports PROV-O, PROV-N, PROV-XML, PROV-JSON)[4], checker.pl (only supports PROV-XML)[5]. As already stated in Section 3.3, RDF serializations of the exported provenance are of significant importance for this thesis. Hence, checker.pl is not the optimal choice to justify the argument that the extensions to YesWorkflow and noWorkflow are PROV compliant.

Since prov-check can be executed locally, the following paragraphs demonstrate the claimed standard compliance by employing prov-check. The interested reader is encouraged to do the same with Prov-Validator, either using its web interface or the open REST API. prov-check is a python script that takes Turtle files as input and outputs either `True` (*valid*) or `False` (*invalid*). In the case of invalid provenance data, the category of the constraint violation is displayed, too.

The three sample scripts are constructed such that – with the corresponding command line arguments – every provenance record that might be exported by the extended versions of YesWorkflow and noWorkflow is actually exported at least once (given there are changes in environment conditions and module dependencies). Thus, the sample Turtle files published to GitHub[6] – in combination with a positive prov-check evaluation – provide sufficient evidence that the new software modules are PROV compliant.

Appendix C shows a Listing that automates the task of validating these files. It assumes a directory structure like in the ProvCaptPyEnvs repository. Essentially, there are three crucial segments of code. In lines 5–20, prov-check is either located locally or downloaded from GitHub. Lines 24–26 determine all Turtle files to be validated. Since the validation process may take some minutes for files larger than 500 Kilobytes, a size limit can be specified. Finally, lines 32–38 invoke prov-check with each of the files identified.

Listing 6.1 depicts a shortened version of the output generated by the script from Appendix C. As can be observed, prov-check could be located locally and thus no download was required. Furthermore, the validation process was successful because for

[3] https://github.com/pgroth/prov-check
[4] https://openprovenance.org/services/view/validator (formerly available at https://provenance.ecs.soton.ac.uk/)
[5] https://github.com/jamescheney/prov-constraints
[6] https://github.com/raffaelfoidl/ProvCaptPyEnvs/tree/master/sample_data

each file, prov-check reported `True`. The entire output is available on GitHub[7].

```
$ ./validate_prov-check.sh

Found prov-check at ./prov-check-master/provcheck/provconstraints.py
  No download necessary

Start validation process...
  checking files from ../sample_data
[...]
3/21 (32 kB): ../sample_data/now_L_provo_definitions.ttl True
4/21 (49 kB): ../sample_data/now_L_provo_environment.ttl True
[...]
11/21 (2 kB): ../sample_data/now_M_provo_fileaccesses.ttl True
12/21 (418 kB): ../sample_data/now_M_provo_modules.ttl True
[...]
14/21 (1629 kB): ../sample_data/now_S_provo_activations.ttl True
15/21 (24 kB): ../sample_data/now_S_provo_definitions.ttl True
19/21 (11 kB): ../sample_data/yw_L.ttl True
20/21 (4 kB): ../sample_data/yw_M.ttl True
21/21 (4 kB): ../sample_data/yw_S.ttl True
Done.
```

Listing 6.1: Result of Validation of Sample Data using prov-check

This result provides evidence to conclude that the features implemented for YesWorkflow and noWorkflow are indeed in compliance with PROV-DM and PROV-CONSTRAINTS.

6.2 Practical Applications of the PROV-O Representation

After the preceding section established the validity of the new provenance export modules, this section seeks to point out how YesWorkflow's and noWorkflow's new output encodings might be utilized more effectively than it was possible before. At first, it will be shown how to answer common questions when analyzing a workflow (execution). Afterwards, the provenance collected during two noWorkflow trials will be used to discover a bug in a script that was presumably introduced during refactoring operations.

6.2.1 Analyzing Workflow Specifications and Script Executions

This subsection presents queries that have been conceived to leverage PROV-O data. In order not to waste space, only three of them can be directly shown in the work at hand. The remaining 18 queries are available at GitHub[8] and may be experimented with using

[7] https://github.com/raffaelfoidl/ProvCaptPyEnvs/tree/master/validate
[8] https://github.com/raffaelfoidl/ProvCaptPyEnvs/tree/master/queries

the SPARQL Playground[9]. For the same reason, namespace prefixes are omitted in the queries – the SPARQL Playground is preconfigured to automatically prepend the required prefixes. Note that the queries have been designed with the principle *clarity over brevity* in mind. That means, there probably are shorter or more elegant ways to formulate them and arrive at the same results, but that could possibly obscure their original intent.

All queries referenced in the following paragraphs are listed in Appendix D. The sample files which are used as exemplary input data are available on GitHub[10].

YesWorkflow: Determine a Workflow's Results

A very fundamental piece of knowledge about a workflow is the data it produces, i. e. its return values. The query in Listing D-1 (number 4 in SPARQL Playground) selects entities that are generated by some activity, but never consumed. Such data typically – but not necessarily – denotes overall outputs of a workflow.

In lines 2–4, all return values of functions are selected. Subsequently, lines 6–9 restrict the query result to those values that are not consumed by any other function. Table 6.2 exemplifies an evaluation of this query with `yw_L.ttl` as input.

return_value	function
yw:conflict_report	yw:persist_conflict_report
yw:frequency_plot	yw:provide_data_info
yw:overall_resource_accesses	yw:get_overlaps
yw:res_summary_csv	yw:persist_res_summary
yw:res_summary_plot	yw:persist_res_summary

Table 6.2: Result of SPARQL Query from Listing D-1

This result indicates that the workflow represented by the large test script exhibits five output parameters, two of which – `res_summary_csv` and `res_summary_plot` – are generated by the same function `persist_res_summary`.

YesWorkflow: Retrieve Functions that influence Return Values

A slightly more interesting problem is determining all functions that may affect a workflow's output parameter(s). This can be solved by the query in Listing D-2 (number 6 in SPARQL Playground). The query itself is particularly intriguing because it includes *property paths*. This feature has been introduced with the specification of SPARQL 1.1 and allows for a much more versatile navigation through RDF graphs. Definitions and examples of their syntax are presented in the official specification [HS13]. For more detailed explanations, one might consult related literature, such as [DuC13, pp. 63–68].

In order to comprehend this query, it may be helpful to make oneself familiar with the structure of Turtle files generated by YesWorkflow. For each return value of a function,

[9] https://github.com/raffaelfoidl/sparql-playground
[10] https://github.com/raffaelfoidl/ProvCaptPyEnvs/tree/master/sample_data

a `qualifiedGeneration` statement is produced (line 5). By means of this qualified generation, the generating activity can be determined. This activity may also use entities as input parameters which are again expressed as a `qualifiedGeneration`. Such a recurring path leading from a return value of an activity to its input parameter(s) is repeatedly followed (denoted by a $*$ quantifier) until an activity without input parameters is reached (line 12). When `yw_M.ttl` is used as input file, this query returns the information depicted by Table 6.3.

return_value	affected_by	functions
yw:blurred_image	2	apply_gauss, read_input
yw:diff_file	4	apply_gauss, save_diff_image, calculate_differences, read_input
yw:diff_image	3	apply_gauss calculate_differences, read_input
yw:input_file	1	read_input
yw:out_file	3	apply_gauss, read_input, save_out_file

Table 6.3: Result of SPARQL Query from Listing D-2

Examining this query evaluation, it can be concluded that there are 4 functions which may have an impact on the workflow output `diff_file`; the same applies to the remaining table rows. These findings might also be better comprehended when examining them in combination with a visualization of `yw_M.ttl`[11].

noWorkflow: Get Activations of a specific Function

For specific use cases, it might be necessary to analyze the concrete return values of functions with respect to their inputs, or simply the runtime of an operation. These questions can be answered by the query shown in Listing D-3 (number 16 in SPARQL Playground) in combination with profiling data collected by noWorkflow.

Lines 3–7 state that function activations of `get_classification` should be selected. Moreover, neither return values (lines 9–16) nor arguments (lines 18–27) are compulsory. This is due to the fact that functions which return a value, but do not exhibit any arguments (or vice versa) should be included in the result as well. In order to achieve a compact yet informative result table, a grouping by the function activation and the activating line in the source code is performed (line 30). Ultimately, the slightly lengthy SELECT clause concatenates the arguments of each function in a readable way (line 1). An evaluation of this query with `now_S_provo_activations.ttl` as input data file is shown in Table 6.4.

[11] https://github.com/raffaelfoidl/ProvCaptPyEnvs/blob/master/yesworkflow_outputs/medium/inst_m_pdf.pdf

L	start_time	end_time	args	return_value
87	14:36:35.45860	14:36:35.45863	{SPEED=118.3}	(11,'Violent Storm')
87	14:36:35.45872	14:36:35.45874	{SPEED=115.0}	(11,'Violent Storm')
87	14:36:35.45882	14:36:35.45885	{SPEED=114.9}	(11,'Violent Storm')
87	14:36:35.45894	14:36:35.45896	{SPEED=104.4}	(11,'Violent Storm')
105	14:36:35.47397	14:36:35.47402	{SPEED=129.6}	(12,'Hurricane')
105	14:36:35.47476	14:36:35.47481	{SPEED=123.6}	(12,'Hurricane')
105	14:36:35.47502	14:36:35.47506	{SPEED=119.3}	(12,'Hurricane')
105	14:36:35.47517	14:36:35.47519	{SPEED=118.8}	(12,'Hurricane')

Table 6.4: Parts of Result of SPARQL Query from Listing D-3

In order to make the table less wide, one column has been removed, another one renamed and date-times have been reduced to their time component. Furthermore, owing to their high degree of similarity, only eight of the twenty activations found are depicted.

Nevertheless, it may be deduced from the first row that the function `get_classification` was activated by line 87 for a duration of `0.00003` seconds and categorized a wind speed of `118.3` as `Violent Storm`, equivalent to a Beaufort number of `11`.

6.2.2 Case Study: Discovering a Semantic Error in a Script

It is assumed that the small sample script was refactored recently. However, latest executions indicate that these modifications introduced a bug in the classification of wind speed measurements. The query from Listing D-4 (number 22 in SPARQL Playground) is employed to retrospectively analyze the classification process. It is derived from the already discussed query from Listing D-3. As this version is customized to the function `get_classification` instead of being as general as possible and much simpler, no further explanations regarding its structure are required. Tables 6.5 and 6.6 are a juxtaposition of correct (left-hand side) and incorrect (right-hand side) results of the script.

speed	classification
18.6	(3,'Gentle breeze')
76.9	(9,'Strong or severe gale')
129.4	(12,'Hurricane')
113.6	(11,'Violent Storm')
32.0	(5,'Fresh breeze')
17.3	(3,'Gentle breeze')

Table 6.5: Classifications before Refactoring

speed	classification
18.6	(2,'Light breeze')
76.9	(2,'Light breeze')
129.4	(11'Violent Storm')
113.6	(11,'Violent Storm')
32.0	(2,'Light breeze')
17.3	(2,'Light breeze')

Table 6.6: Classifications after Refactoring

Table 6.6 clearly demonstrates faulty behavior, but no immediately obvious reason. Nevertheless, errors when reading and interpreting the script's input data can be ruled out. This is justified because `get_classification` consumes the same, correct, input

parameters in both trials. Hence, it may be concluded that the bug is caused by the changes made to this function.

noWorkflow's `restore` command allows for a convenient retrieval of the script's contents at the time of the two trials. Listing 6.2 shows a sequence of commands that write the original and the refactored script to respective files as well as generate a line-by-line comparison of them. On some systems, the `--suppress-common-lines` option of the diff utility might not be available. In such cases, the output also contains lines that do not differ between the two files.

```
now restore 1 -f inst_s.py -t inst_s_original.py
now restore 2 -f inst_s.py -t inst_s_refactored.py
diff -y --suppress-common-lines inst_s_original.py inst_s_refactored.py
```

Listing 6.2: Comparison of the original and refactored script

Figure 6.1 shows a colored version of the comparison report generated by diff. It reveals that the goal was to eliminate redundant upper bounds because they must coincide with the lower bounds of the next higher classes.

```
47      global root                                          47      infinity = float("inf")
48      classes = {                                          48      classes = {
49          (0, 0., 1.85): "Calm",                           49          (0, 0.): "Calm",
50          (1, 1.85, 7.41): "Light Airs",                   50          (1, 1.85): "Light Airs",
51          (2, 7.41, 12.96): "Light breeze",                51          (2, 7.41): "Light breeze",
52          (3, 12.96, 20.37): "Gentle breeze",              52          (3, 12.96): "Gentle breeze",
53          (4, 20.37, 29.63): "Moderate breeze",            53          (4, 20.37): "Moderate breeze",
54          (5, 29.63, 40.74): "Fresh breeze",               54          (5, 29.63): "Fresh breeze",
55          (6, 40.74, 51.86): "Strong breeze",              55          (6, 40.74): "Strong breeze",
56          (7, 51.86, 62.97): "Moderate (near) gale",       56          (7, 51.86): "Moderate (near) gale",
57          (8, 62.97, 75.93): "Fresh gale",                 57          (8, 62.97): "Fresh gale",
58          (9, 75.93, 88.9): "Strong or severe gale",       58          (9, 75.93): "Strong or severe gale",
59          (10, 88.9, 103.71): "Whole gale or Storm",       59          (10, 88.9): "Whole gale or Storm",
60          (11, 103.71, 118.53): "Violent Storm",           60          (11, 103.71): "Violent Storm",
61          (12, 118.53, 118.54): "Hurricane"                61          (12, 118.53): "Hurricane",
                                                             62          (12, infinity): "Hurricane"
62      }                                                    63      }
63                                                           64
64      speed = min(118.54, speed)                           65      items = list(classes.keys())
65      for level, lower, upper in classes:                  66      for i, (level, lower) in enumerate(items[:-1]):
66          if lower < speed <= upper:                       67          _, lower_next = items[i + 1]
67              return level, classes[(level, lower, upper)] 68          if lower < speed <= lower_next:
                                                             69              return level, classes[(level, lower)]
```

Figure 6.1: Changes made to `get_classification` during Refactoring

This approach is not guaranteed to be correct – as Table 6.6 has already established. It assumes that the order when iterating over the dictionary `classes` is not only consistent, but also congruent with the insertion order. Whether this assumption is true is highly dependent on the Python version and implementation (e. g. CPython, Jython). Hence, one should generally not rely on it, unless the Python environment is known and not subject to change. In order to make the refactoring valid, another data structure could be used that avoids the dictionary-induced indeterminism.

Figure 6.2 exemplifies this utilizing OrderedDict, a dict subclass that guarantees iteration in insertion order.

```
48    classes = {
49        (0, 0.): "Calm",
50        (1, 1.85): "Light Airs",
51        (2, 7.41): "Light breeze",
52        (3, 12.96): "Gentle breeze",
53        (4, 20.37): "Moderate breeze",
54        (5, 29.63): "Fresh breeze",
55        (6, 40.74): "Strong breeze",
56        (7, 51.86): "Moderate (near) gale",
57        (8, 62.97): "Fresh gale",
58        (9, 75.93): "Strong or severe gale",
59        (10, 88.9): "Whole gale or Storm",
60        (11, 103.71): "Violent Storm",
61        (12, 118.53): "Hurricane",
62        (12, infinity): "Hurricane"
63    }
```

```
49    classes = OrderedDict([
50        ((0, 0.), "Calm"),
51        ((1, 1.85), "Light Airs"),
52        ((2, 7.41), "Light breeze"),
53        ((3, 12.96), "Gentle breeze"),
54        ((4, 20.37), "Moderate breeze"),
55        ((5, 29.63), "Fresh breeze"),
56        ((6, 40.74), "Strong breeze"),
57        ((7, 51.86), "Moderate (near) gale"),
58        ((8, 62.97), "Fresh gale"),
59        ((9, 75.93), "Strong or severe gale"),
60        ((10, 88.9), "Whole gale or Storm"),
61        ((11, 103.71), "Violent Storm"),
62        ((12, 118.53), "Hurricane"),
63        ((12, infinity), "Hurricane")
64    ])
```

Figure 6.2: Modifications resolving incorrect Behavior of `get_classification`

After applying these changes, `get_classification` returns the correct results from Table 6.5.

The above paragraphs showed how to use different modules of noWorkflow to analyze script executions with the goal of locating a newly introduced programming error. For such cases, the tool is particularly useful because – owing to the provenance collection of trials – it offers a much finer granularity than other mechanisms for discovering differences (e. g. commits in a version control system). Although the example given is admittedly simple, it still demonstrates a general strategy to capitalize on the PROV-O representation of the collected provenance data for diagnostic purposes.

All files related to this case study are available on GitHub[12].

6.3 Comparison of YesWorkflow's and noWorkflow's Provenance

Since both YesWorkflow and noWorkflow capture prospective provenance, it might be interesting to compare these pieces of information. For this purpose, Figure 6.3 shows a visualization of the provenance of the function `persist_top_ten_overall` which was created by YesWorkflow with the small test script as input. Additionally, Listing 6.3 depicts an excerpt of the definition provenance of the same function recorded by noWorkflow in PROV-N notation.

It is immediately evident that the information gathered by YesWorkflow is conceptually simpler. This is consistent with the tool's rationale of providing an abstracted view of scripts that is understandable for humans. By contrast, noWorkflow offers more comprehensive details that are meant for machines or in-depth investigations. As a consequence, the prospective provenance of YesWorkflow and noWorkflow do not necessarily match. For example, line 1 of Listing 6.3 indicates the position of `persist_top_ten_overall`

[12] https://github.com/raffaelfoidl/ProvCaptPyEnvs/tree/master/case_study

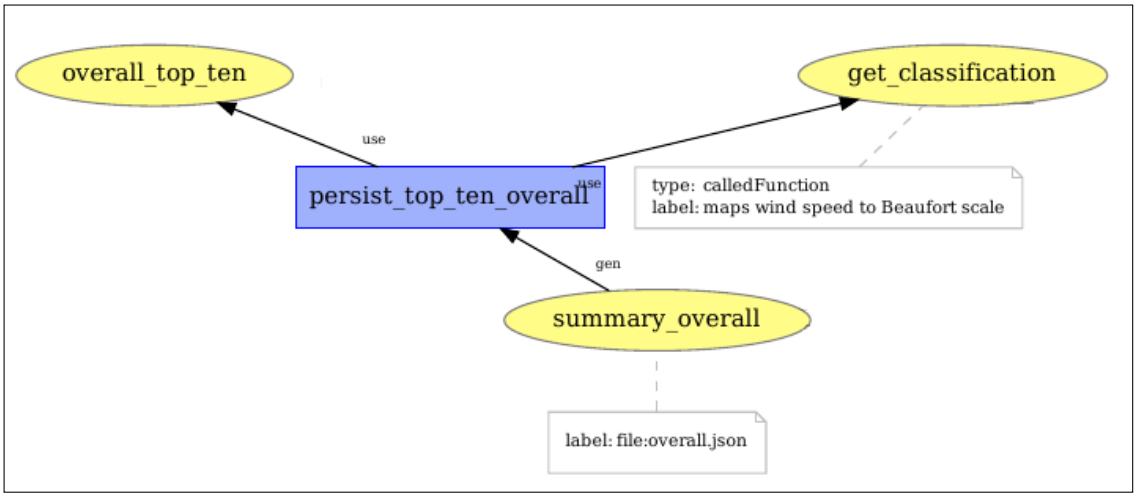

Figure 6.3: Visualization of the Provenance of a Function captured by YesWorkflow

```
1  activity(functionDefinition7, −, −, [lastLine=111, firstLine=101,
       ↪ codeHash="014f08ecdfb0b6e96fa623bec3c75b49566ec54c",
       ↪ prov:type="functionDefinition", prov:label="persist_top_ten_overall"])
2
3  entity(argumentDefinition38,[prov:label="data",prov:type="argumentDefinition"])
4  entity(argumentDefinition39,[prov:label="path",prov:type="argumentDefinition"])
5  used(funcDef7UsedArgDef38; functionDefinition7, argumentDefinition38, −,
       ↪ [prov:role="argument", prov:type="argumentDefinition"])
6  used(funcDef7UsedArgDef39; functionDefinition7, argumentDefinition39, −,
       ↪ [prov:role="argument", prov:type="argumentDefinition"])
7
8  activity(callDefinition40, −, −, [prov:label="enumerate",
       ↪ prov:type="callDefinition"])
9  activity(callDefinition41, −, −, [prov:label="get_classification",
       ↪ prov:type="callDefinition"])
10 [...]
11 activity(callDefinition46, −, −, [prov:label="json.dump",
       ↪ prov:type="callDefinition"])
12
13 wasInformedBy(callDef40CalledByFuncDef7; callDefinition40, functionDefinition7,
       ↪  [prov:type="callDefinition"])
14 wasInformedBy(callDef41CalledByFuncDef7; callDefinition41, functionDefinition7,
       ↪  [prov:type="callDefinition"])
15 [...]
16 wasInformedBy(callDef46CalledByFuncDef7; callDefinition46, functionDefinition7,
       ↪  [prov:type="callDefinition"])
```

Listing 6.3: noWorkflow's Definition Provenance of the Function from Figure 6.3

in the script's source code – this information is not present in Figure 6.3. Lines 3 and 4 further show a discrepancy between the two PROV-O data sets: While Listing 6.3 states the parameters `data` and `path` as they are declared in the script, Figure 6.3 displays their names as defined by the person who authored the YesWorkflow annotations (the parameter `path` is, in fact, even hidden). Depending on the accuracy of the annotations, such inconsistencies may also occur with the names of functions and function calls.

6.4 Summary

This chapter provided evidence for the claim that the new export modules for YesWorkflow and noWorkflow indeed comply with the PROV-O standard. Moreover, it was shown how to query this PROV-O data using SPARQL in order to cover different use cases such as analyzing a workflow (execution) or examining the behavior of a function across different trials. Finally, a brief comparison of the structure of definition provenance captured by YesWorkflow and noWorkflow was given.

CHAPTER 7

Conclusion

7.1 Discussion

This thesis explored how the utility of existing approaches to capturing provenance in Python environments can be increased. The goal was achieved by extending two selected solutions – YesWorkflow and noWorkflow – such that they produce output which is compliant to the World Wide Web Consortium's PROV standard. This was motivated by the fact that their lack of a standardized provenance model impedes machine-aided processing and exchanging of their collected data. When utilizing well-defined standards, one can benefit from the programs implementing them.

In the case of the work at hand, RDF encodings – more specifically, Turtle serializations – of the PROV-O data are employed extensively. The Turtle format proves especially useful because it exhibits a high syntactic resemblance to SPARQL. This greatly facilitates the process of designing queries that are meant to infer information from the provenance exported by YesWorkflow and noWorkflow which, until now, was not explicitly available. Combining the strengths of SPARQL with the modifications proposed in this thesis, numerous implicit connections between the data records can be discovered. This was demonstrated by elaborating on four different use cases of the captured provenance. For instance, it was shown how to determine all processes that might have an effect on (specific) workflow results. Furthermore, a common scenario of analyzing the provenance of two workflow executions for the purpose of locating a programming error was outlined.

These contributions seek to narrow the gap between provenance as valuable metadata about scientific workflows and Python scripts. Both YesWorkflow's prospective and noWorkflow's retrospective provenance are useful for their respective applications. However, until now, further processing of their data was difficult. The PROV-O implementation proposed in this thesis demonstrates opportunities to increase the ability to accurately document, reason about and exchange the computational processes as well as results of scientific workflows represented by Python scripts.

7.2 Future Work

Although the results of this thesis are overall pleasing, there is room for improvement.

When constructing queries for the provenance created by YesWorkflow, it became apparent that `@log` annotations would have been better modeled analogously to `@call`. That way, no regular expressions in the SPARQL query to retrieve the individual components of the log message would be required. Instead, they could be matched with triple patterns, as it is also done for other types of provenance.

The work regarding noWorkflow was mostly concentrated on the tool's essential outputs. However, it also provides versioning functionality related to the evolution of scripts that has not been fully covered. Furthermore, in the implementation proposed, attribute values (e. g. argument or return values) are truncated to a predefined maximum length. Otherwise, exported provenance files may grow uncontrollably large – e. g. if lists with many elements act as function arguments. Similarly, the output file size as well as runtime might exceed tolerable thresholds owing to functions that are activated with high frequency – e. g. helper functions in a nested loop. There should be developed a better solution regarding attribute values and it would be desirable to be able to exclude specific functions from the provenance export.

Finally, the composition of more complex queries is worth being addressed. The ones conceived in the course of this thesis are primarily for demonstration purposes and thus have a relatively isolated scope. Hence, future work should include investigations on how to use the provenance captured by YesWorkflow and noWorkflow in order to resolve more sophisticated inquiries about a workflow (execution).

List of Figures

List of Tables

List of Listings

List of Algorithms

Appendix

Appendix A

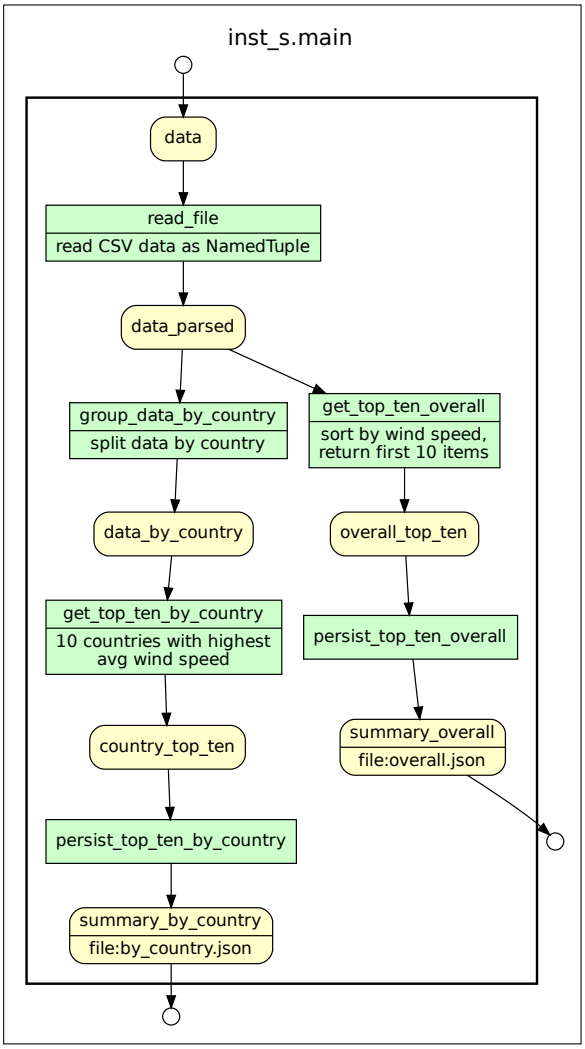

Figure A: Output Image generated by YesWorkflow from Small Example Script

Appendix B

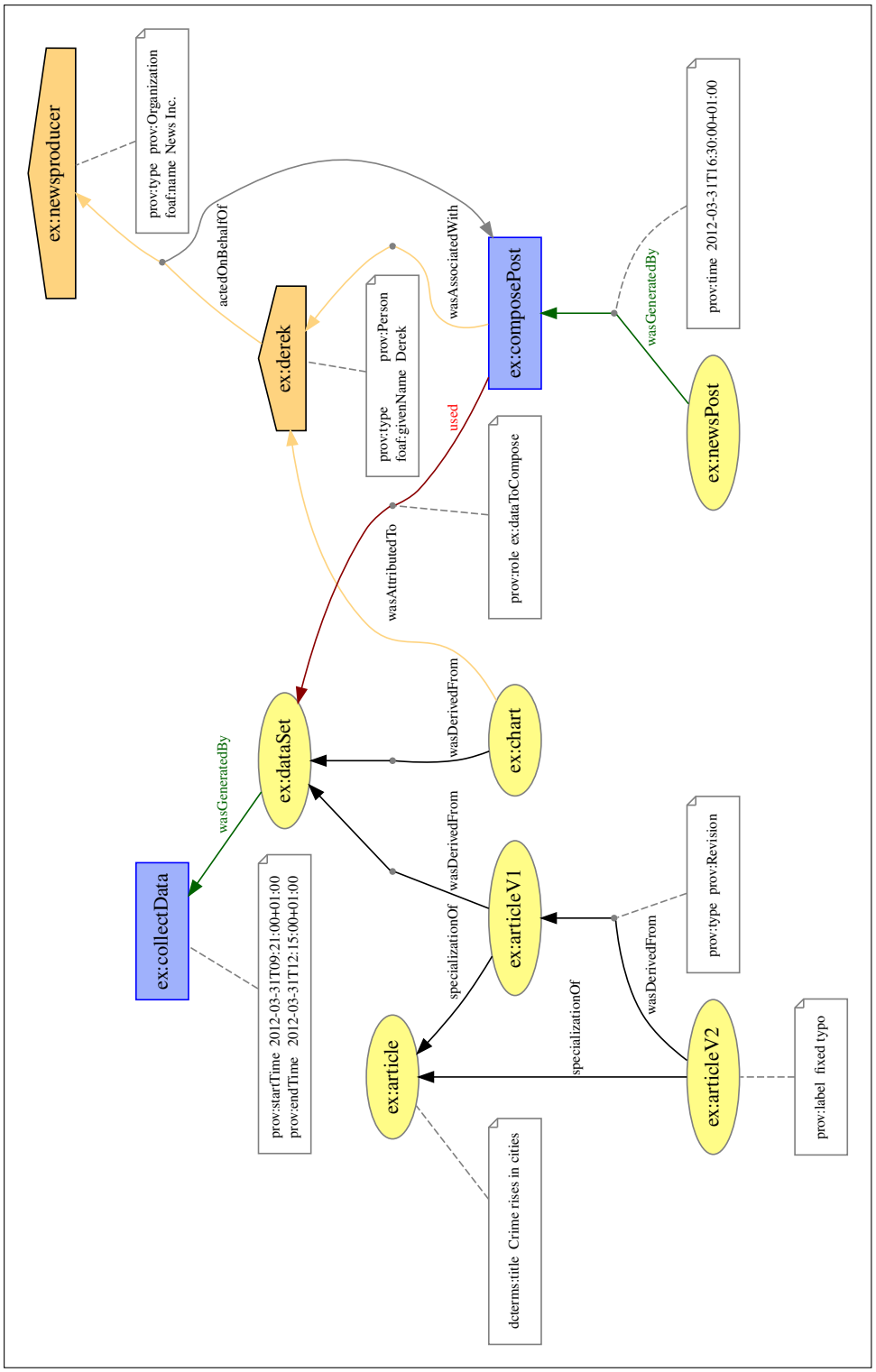

Figure B: Visualization of the PROV-O Document from Listing 3.1

Appendix C

```
#!/bin/bash
CHECKER="./prov-check-master/provcheck/provconstraints.py"

# download prov-check if not found locally
if [ ! -f "$CHECKER" ]; then
 URL="https://github.com/pgroth/prov-check/archive/master.zip"
 echo "Downloading and extracting prov-check..."
 echo " URL: $URL"
 echo ""
 rm -rf prov-check-master
 wget $URL
 unzip master.zip
 rm master.zip
 echo "Done."
 echo ""
else
 echo "Found prov-check at $CHECKER"
 echo " No download necessary"
 echo ""
fi

# specify maximum size of files to be validated
# in order to reduce validation runtime (here: 2048 Kilobytes)
DIR="../sample_data"
FILES=$(find $DIR -type f -size -2048k)
TOTAL=$(ls $FILES | wc -l)

echo "Start validation process..."
echo " checking files from $DIR"
# iterate over files to be validated and print status information
COUNT=1 ; SIZE=0
for file in $FILES
do
 ((SIZE=$(wc -c < "$file") / 1000))
 echo -n "$COUNT/$TOTAL ($SIZE kB): "
 python $CHECKER $file
 ((COUNT+=1))
done
echo "Done."
```

Listing C: Shell Script to validate PROV-O using prov-check

Appendix D

```
SELECT (?entity AS ?return_value) (?activity AS ?function) WHERE {
 ?generation a prov:Generation .
 ?generation prov:activity ?activity .
 ?entity prov:qualifiedGeneration ?generation .

 FILTER NOT EXISTS {
  ?usage a prov:Usage .
  ?usage prov:entity ?entity .
  ?activity2 prov:qualifiedUsage ?usage .
 }
 # optional: exclude certain functions from output computation
 FILTER ( (?activity != yw:inst_l_main) )
}
ORDER BY ?entity ?activity
```

Listing D-1: SPARQL Query for a Script's Output Parameters in YesWorkflow

```
SELECT ?return_value (COUNT(DISTINCT ?function_name) AS ?affected_by)
    ↪ (GROUP_CONCAT(DISTINCT ?function_name; separator=", ") AS ?functions)
WHERE {
 # starting point: ?return_value (could be a hard-coded literal if a specific
 # parameter is of special interest) is generated by some qualified generation
 ?return_value prov:qualifiedGeneration ?generation .

 # from there on, this generation is caused by an activity, which uses
 # entites as parameters which in turn are again generated by some activity.
 # the property path in parenthesis equals a "step from one activity to
 # the next" (traversing the graph "upwards"), until the last one is reached
 # (i. e. one that does not have any inputs)
 ?generation (prov:activity/prov:qualifiedUsage/prov:entity/prov:
    ↪ qualifiedGeneration)*/prov:activity ?activity .

 # optional: exclude certain functions from output computation
 FILTER ( (?activity != yw:inst_m_main) )

 # optional: only display local name instead of full function URI
 BIND(replace(str(?activity), ".*\\/", "") AS ?function_name)
}
GROUP BY ?return_value
ORDER BY ?return_value
```

Listing D-2: SPARQL Query for Functions affecting Return Values in YesWorkflow

```
SELECT ?activation ?L ?start_time ?end_time (GROUP_CONCAT(
    ↪ CONCAT("{", UCASE(?argument_name), " = ", ?argument_value, "}");
    ↪ separator = ", ") AS ?args) ?return_value
WHERE {
 ?activation a prov:Activity, "functionActivation" ;
             rdfs:label "get_classification" ;
             prov:startedAtTime ?start_time ;
             prov:endedAtTime ?end_time ;
             ns1:line ?L .

 OPTIONAL {
  ?generation a prov:Generation ;
              prov:activity ?activation .

  ?retVal a prov:Entity, "returnValue" ;
          prov:qualifiedGeneration ?generation ;
          prov:value ?return_value .
 }

 OPTIONAL {
  ?activation prov:qualifiedUsage ?usage .
  ?usage a prov:Usage, "argumentActivation" ;
         prov:hadRole "argument" ;
         prov:entity ?argumentAct .

  ?argumentAct a prov:Entity, "argumentActivation" ;
               rdfs:label ?argument_name ;
               prov:value ?argument_value .
 }

}
GROUP BY ?activation ?L ?start_time ?end_time ?return_value
ORDER BY ASC(xsd:dateTime(?start_time))
```

Listing D-3: SPARQL Query to retrieve Activations of a specific Function

```
SELECT ?speed ?classification
WHERE {
 ?activation a prov:Activity, "functionActivation" ;
             rdfs:label "get_classification" .

 ?generation a prov:Generation ;
             prov:activity ?activation .

 ?retVal a prov:Entity, "returnValue" ;
         prov:qualifiedGeneration ?generation ;
         prov:value ?classification .

 ?activation prov:qualifiedUsage ?usage .
 ?usage a prov:Usage, "argumentActivation" ;
        prov:hadRole "argument" ;
        prov:entity ?argumentAct .

 ?argumentAct a prov:Entity, "argumentActivation" ;
              prov:value ?speed .
}
GROUP BY ?classification ?speed
```

Listing D-4: SPARQL Query for Wind Speed Classifications in noWorkflow

Bibliography

[ABJF06] Ilkay Altintas, Oscar Barney, and Efrat Jaeger-Frank. Provenance Collection Support in the Kepler Scientific Workflow System. In Luc Moreau and Ian Foster, editors, *Provenance and Annotation of Data (IPAW)*, volume 4145 of *Lecture Notes in Computer Science*, pages 118–132. Springer Berlin Heidelberg, May 2006.

[BCC+05] Louis Bavoil, Steven P. Callahan, Patricia J. Crossno, Juliana Freire, Carlos Scheidegger, Cláudio Silva, and Huy T. Vo. VisTrails: Enabling Interactive Multiple-View Visualizations. In *IEEE Visualization*, pages 135–142. Institute of Electrical and Electronics Engineers, October 2005.

[BGS08] Carsten Bochner, Roland Gude, and Andreas Schreiber. A Python Library for Provenance Recording and Querying. In Juliana Freire, David Koop, and Luc Moreau, editors, *Provenance and Annotation of Data and Processes (IPAW)*, volume 5272 of *Lecture Notes in Computer Science*, pages 229–240. Springer Berlin Heidelberg, June 2008.

[CMMDN13] James Cheney, Paolo Missier, Luc Moreau, and Tom De Nies. Constraints of the PROV Data Model. W3C Recommendation, World Wide Web Consortium, April 2013. http://www.w3.org/TR/2013/REC-prov-constraints-20130430/. Accessed on 2020-03-20.

[CWL14] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, World Wide Web Consortium, February 2014. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/. Accessed on 2020-03-15.

[DBK+15] Saumen Dey, Khalid Belhajjame, David Koop, Meghan Raul, and Bertram Ludäscher. Linking Prospective and Retrospective Provenance in Scripts. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP)*. USENIX Association, July 2015.

[DF08] Susan B. Davidson and Juliana Freire. Provenance and Scientific Workflows: Challenges and Opportunities. In *ACM SIGMOD International Conference on Management of Data*, SIGMOD 2008, page 1345–1350. Association for Computing Machinery, June 2008.

[DuC13] Bob DuCharme. *Learning SPARQL: Querying and Updating with SPARQL 1.1*. O'Reilly Media, July 2013.

[GAVS11] Stephan Grimm, Andreas Abecker, Johanna Völker, and Rudi Studer. Ontologies and the Semantic Web. In John Domingue, Dieter Fensel, and James A. Hendler, editors, *Handbook of Semantic Web Technologies*, pages 507–579. Springer Berlin Heidelberg, May 2011.

[GKHT11] Fabien L. Gandon, Reto Krummenacher, Sung-Kook Han, and Ioan Toma. Semantic Annotation and Retrieval: RDF. In John Domingue, Dieter Fensel, and James A. Hendler, editors, *Handbook of Semantic Web Technologies*, pages 117–155. Springer Berlin Heidelberg, May 2011.

[GM13] Paul Groth and Luc Moreau. PROV-Overview. W3C Note, World Wide Web Consortium, April 2013. http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/. Accessed on 2020-03-14.

[GMB+13] Yolanda Gil, Simon Miles, Khalid Belhajjame, Helena Deus, Daniel Garijo, Graham Klyne, Paolo Missier, Stian Soiland-Reyes, and Stephan Zednik. PROV Model Primer. W3C Note, World Wide Web Consortium, April 2013. http://www.w3.org/TR/2013/NOTE-prov-primer-20130430/. Accessed on 2020-03-14.

[GS12] Philip J. Guo and Margo Seltzer. BURRITO: Wrapping Your Lab Notebook in Computational Infrastructure. In *4th USENIX Workshop on the Theory and Practice of Provenance (TaPP)*. USENIX Association, June 2012.

[HAW13] Mohammad Rezwanul Huq, Peter M. G. Apers, and Andreas Wombacher. ProvenanceCurious: A Tool to Infer Data Provenance from Scripts. In *Conference on Extending Database Technology (EDBT)*, pages 765–768. Association for Computing Machinery, March 2013.

[HS13] Steven Harris and Andy Seaborne. SPARQL 1.1 Query Language. W3C Recommendation, World Wide Web Consortium, March 2013. http://www.w3.org/TR/2013/REC-sparql11-query-20130321/. Accessed on 2020-03-15.

[LAB+06] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, December 2006.

[LLCF10] Chunhyeok Lim, Shiyong Lu, Artem Chebotko, and Farshad Fotouhi. Prospective and Retrospective Provenance Collection in Scientific Workflow Environments. In *IEEE International Conference on Services Computing (SCC)*, pages 449–456. Institute of Electrical and Electronics Engineers, July 2010.

[LSM+13] Timothy Lebo, Satya Sahoo, Deborah McGuinness, Khalid Belhajjame, James Cheney, David Corsar, Daniel Garijo, Stian Soiland-Reyes, Stephan Zednik, and Jun Zhao. PROV-O: The PROV Ontology. W3C Recommendation, World Wide Web Consortium, April 2013. http://www.w3.org/TR/2013/REC-prov-o-20130430/. Accessed on 2020-03-14.

[MBBL15] Timothy McPhillips, Shawn Bowers, Khalid Belhajjame, and Bertram Ludäscher. Retrospective Provenance Without a Runtime Provenance Recorder. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP)*. USENIX Association, July 2015.

[MBC+14] Leonardo Murta, Vanessa Braganholo, Fernando Chirigati, David Koop, and Juliana Freire. noWorkflow: Capturing and Analyzing Provenance of Scripts. In Bertram Ludäscher and Beth Plale, editors, *Provenance and Annotation of Data and Processes (IPAW)*, volume 8628 of *Lecture Notes in Computer Science*, pages 71–83. Springer International Publishing, June 2014.

[MBK+09] Pierre Mouallem, Roselyne Barreto, Scott Klasky, Norbert Podhorszki, and Mladen Vouk. Tracking Files in the Kepler Provenance Framework. In Marianne Winslett, editor, *Scientific and Statistical Database Management (SSDBM)*, volume 5566 of *Lecture Notes in Computer Science*, pages 273–282. Springer Berlin Heidelberg, June 2009.

[MG13] Luc Moreau and Paul Groth. *Provenance: An Introduction to PROV*. Number 7 in Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan & Claypool Publishers, August 2013.

[MMB+13] Luc Moreau, Paolo Missier, Khalid Belhajjame, Reza B'Far, James Cheney, Sam Coppens, Stephen Cresswell, Yolanda Gil, Paul Groth, Graham Klyne, Timothy Lebo, Jim McCusker, Simon Miles, James Myers, Satya Sahoo, and Curt Tilmes. PROV-DM: The PROV Data Model. W3C Recommendation, World Wide Web Consortium, April 2013. http://www.w3.org/TR/2013/REC-prov-dm-20130430/. Accessed on 2020-03-14.

[MMCSR13] Luc Moreau, Paolo Missier, James Cheney, and Stian Soiland-Reyes. PROV-N: The Provenance Notation. W3C Recommendation, World Wide Web Consortium, April 2013. http://www.w3.org/TR/2013/REC-prov-n-20130430/. Accessed on 2020-03-14.

[MSK+15] Timothy McPhillips, Tianhong Song, Tyler Kolisnik, Steve Aulenbach, Khalid Belhajjame, R. Kyle Bocinsky, Yang Cao, James Cheney, Fernando Chirigati, Saumen Dey, Juliana Freire, Christopher Jones, James Hanken, Keith W. Kintigh, Timothy A. Kohler, David Koop, James A. Macklin, Paolo Missier, Mark Schildhauer, Christopher Schwalm, Yaxing Wei, Mark Bieda, and Bertram Ludäscher. YesWorkflow: A User-Oriented,

Language-Independent Tool for Recovering Workflow Information from Scripts. *International Journal of Digital Curation*, 10:298–313, February 2015.

[MSRO+10] Paolo Missier, Stian Soiland-Reyes, Stuart Owen, Wei Tan, Alexandra Nenadic, Ian Dunlop, Alan Williams, Tom Oinn, and Carole Goble. Taverna, reloaded. In Michael Gertz and Bertram Ludäscher, editors, *Scientific and Statistical Database Management (SSDBM)*, volume 6187 of *Lecture Notes in Computer Science*, pages 471–481. Springer Berlin Heidelberg, July 2010.

[OAF+04] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, and Peter Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, June 2004.

[PBMF15] João Felipe Nicolaci Pimentel, Vanessa Braganholo, Leonardo Murta, and Juliana Freire. Collecting and Analyzing Provenance on Interactive Notebooks: When IPython Meets noWorkflow. In *7th USENIX Workshop on the Theory and Practice of Provenance (TaPP)*. USENIX Association, July 2015.

[PDM+16] João Felipe Pimentel, Saumen Dey, Timothy McPhillips, Khalid Belhajjame, David Koop, Leonardo Murta, Vanessa Braganholo, and Bertram Ludäscher. Yin & Yang: Demonstrating Complementary Provenance from noWorkflow & YesWorkflow. In Marta Mattoso and Boris Glavic, editors, *Provenance and Annotation of Data and Processes (IPAW)*, pages 161–165. Springer International Publishing, June 2016.

[PFBM16] João Felipe Pimentel, Juliana Freire, Vanessa Braganholo, and Leonardo Murta. Tracking and Analyzing the Evolution of Provenance from Scripts. In Marta Mattoso and Boris Glavic, editors, *Provenance and Annotation of Data and Processes (IPAW)*, pages 16–28. Springer International Publishing, June 2016.

[PMBF17] João Felipe Pimentel, Leonardo Murta, Vanessa Braganholo, and Juliana Freire. noWorkflow: a Tool for Collecting, Analyzing, and Managing Provenance from Python Scripts. *International Conference on Very Large Data Bases (VLDB)*, 10:1841–1844, August 2017.

[SR14] Guus Schreiber and Yves Raimond. RDF 1.1 Primer. W3C Note, World Wide Web Consortium, June 2014. http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/. Accessed on 2020-03-15.

[The17] The Editors of Encyclopaedia Britannica. Beaufort scale. Encyclopaedia Britannica, March 2017. https://www.britannica.com/science/Beaufort-scale. Accessed on 2020-03-16.

[TSWH05] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. Visual Grid Workflow in Triana. *Journal of Grid Computing*, 3(3-4):153–169, September 2005.

[ZCW+18] Qian Zhang, Yang Cao, Qiwen Wang, Duc Vu, Priyaa Thavasimani, Timothy McPhillips, Paolo Missier, Peter Slaughter, Christopher Jones, Mathew B. Jones, and Bertram Ludäscher. Revealing the Detailed Lineage of Script Outputs Using Hybrid Provenance. *International Journal of Digital Curation*, 12:390–408, August 2018.

[ZGH13] Stephan Zednik, Paul Groth, and Trung Dong Huynh. PROV Implementation Report. W3C Note, April 2013. http://www.w3.org/TR/2013/NOTE-prov-implementations-20130430/. Accessed on 2020-03-20.