FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Reproduzierbarkeit via ontologischer Darstellung der Provenance

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Richard Roth
Matrikelnummer 1429045

an der Fakultät für Informatik

der Technischen Universität Wien

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

Wien, 1. Jänner 2018

Richard Roth	Andreas Rauber

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Reproducibility by Ontological Representation of Provenance

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Richard Roth
Registration Number 1429045

to the Faculty of Informatics

at the TU Wien

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

Vienna, 1st January, 2018

Richard Roth   Andreas Rauber

Erklärung zur Verfassung der Arbeit

Richard Roth
Liechtensteinstraße 56/15, 1090 Wien, Österreich

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

Wien, 1. Jänner 2018

Richard Roth

Acknowledgements

To my beautiful Roxy: Thank you for being there every step of the way, for listening while I nerded out about the work in this thesis, and for always having the patience to smile while doing so. To say you have been my rock on this crazy adventure would be an understatement. Thank you for everything you have done, and continue to do, for me, and for us.

To my Austrian family: Thank you for letting me into your lives and giving me a place I can call home. I will be forever grateful for the warmth with which you have all welcomed me into your family, and for having made this huge step in my life easier than I could have ever hoped for.

To my loving mother: I would not be who I am, or where I am, without your unwavering love, emotional support, and moral guidance. Thank you for always encouraging me to follow my heart, and for believing in me even when I didn't believe in myself.

To my brothers: You've all always been there for me in your own individual and goofy ways. Thank you for being my best friends, and for always being a place of comfort that can never be rivaled.

To my late father: You instilled in me a sense a responsibility, pride, and accountability for my actions and my work. I hope that everything I have accomplished up until this point in my life would have made you proud, for none of it would have been possible without you. This one is for you.

I would of course also like to extend my deepest gratitude to my mentor Andreas Rauber for his enthusiasm, support, and vision while working on this thesis.

Kurzfassung

Die Fähigkeit, Ergebnisse zu reproduzieren, welche aus Computerrecherche resultieren, ist unerlässlich für den Erfolg und die Glaubwürdigkeit des Faches. Diese wird jedoch durch zunehmend größer werdende Datensätze und verschiedenster Kombinationen von Hardware und Software Umgebungen deutlich erschwert. Bereits existierende Lösungen fokussieren darauf Workflows erneut auszuführen, durch welche die ursprünglichen Ergebnisse erzielt wurden. Allerdings sind diese Lösungen begrenzt, da sie sich nur mit limitierten Prozessausführungen beschäftigen. Diese Arbeit befasst sich damit dieses Problem durch die kontinuierliche und autonome Erfassung der systemweiten Provenance zu beheben und dies in einer ontologischen Form darzustellen. Auf diese Weise, implementieren wir eine vollständige Systemlösung mit welcher Benutzer ihre Recherchen ausführen können, ohne den zusätzlichen Mehraufwand der manuellen Verfolgung und Manipulation dieser Informationen in Kauf nehmen zu müssen. In dieser wissenschaftlichen Arbeit, beschreiben wir unseren Lösungsvorschlag dies zu erzielen und die Technologien die ihn umfassen. Zusätzlich veranschaulichen wir zwei Methoden mittels welcher Benutzer die resultierende Ontologie visualisieren und mit ihr interagieren können. Schlussendlich, wird untersucht inwieweit unsere vorgeschlagene Lösung die erforderlichen Informationen bereitstellt um die Reproduzierbarkeit von Ergebnissen der Computerrecherche sowie ihrer zugehörigen Anwendungen zu ermöglichen.

Abstract

The ability to reproduce results stemming from computational research is paramount to the field's success and its credibility. Making this task difficult are the increasingly large data sets and assorted combinations of hardware and software environments being employed by researchers. Existing solutions focus on re-executing the workflows from which the original results were obtained. However, these solutions are limited in the sense that they address only confined process executions. This work aims to rectify this problem by continuously and autonomously capturing system wide provenance and representing it in ontological form. In doing so, we implement a complete system solution on which users can perform their research without the additional overhead of having to manually track and manipulate this information themselves. In this paper, we explore our proposed solution and the technologies it is comprised of. Additionally, we illustrate two methods in which users can visualize and interact with the resulting ontology. Finally, we examine to what extent our proposed solution provides the necessary information to better enable the reproducibility of computational research results and its viable applications.

Contents

CHAPTER 1

Introduction

1.1 Motivation

With computers becoming progressively more capable, their role and influence in the academic and industry communities continue to expand. The computational power they provide is being increasingly utilized in not only scientific studies and research, but also in critical applications found in a variety of industries such as transportation, communication, and medical care. However, unlike traditional laboratory science, the results of scientific research performed via the application of computational science mechanisms are often difficult, or even impossible, to reproduce. This can be attributed to experiments and applications that depend on specific input data and parameters being executed in a variety of inconsistent software and hardware environments. Often times, the aforementioned experiments and applications lack the proper documentation to assists those who wish to reproduce and build upon the original result(s). In an effort to address this problem, much research has been focused on how to not only create digital work that is inherently more reproducible, but also on developing tools and utilities designed to allow users and researchers and users to reliably validate and redeploy processes. However, reproducing results in this manner is not always possible as a result of often complicated workflows and process' performed on what can best be described as a dynamic medley of hardware and software.

1.2 Problem Statement

Modern buzzwords such as 'provenance' and 'ontology' are regularly found in research pertaining to the issue of reproducibility. Mechanisms employing the concept of provenance share the common goal of tracing a program's execution, including its software dependencies, the hardware and software environment in which it is executed, and its inputs and outputs. This information can then be stored in an ontology, allowing for a concise and structured representation of the relationships that exist between the gathered provenance information. However, while one won't be hard-pressed to find a tool that serves to accomplish either of these goals, still missing is a tool that combines both of these concepts. It is thus highly desirable to develop a solution that not only accomplishes just that, but also does so in an accessible, automated, system-wide manner.

As a concrete example, we take a step back and consider the task of tracing the provenance of a single Personal Document Format (PDF) file. More specifically, we wish to track the provenance of all tables, images, etc., that are found in this PDF. First, we face the challenge of capturing all of the pertinent information. Second, the challenge of composing this information in the structure of an ontology. And lastly, the challenge of presenting this information in a human-interactable form. Furthermore, this should be performed automatically without user interaction, and not only for this single PDF, but on a system-wide scale. To accomplish this task, a solution utilizing an interactive and easy-to-use visual interface must be developed. The ideal tool would allow for users of all skill ranges to quickly and efficiently identify a given component's provenance and to navigate the constructed ontology by means of graphical visualization.

1.3 Aim of this Work

This main result of this thesis is the prototype of a tool allowing the user(s) to view the provenance of a given file on a very granular scale. That is to say, not only will the provenance of a file be known, but also any components of the file such as images, tables, graphs, etc. Users are able to interact with a graphical representation of an ontology that captures the provenance information, and to expand or collapse nodes to the desired level of detail. This solution thus solves the following three problems:

1) How to monitor and capture the provenance of files on a system-wide scale?
2) How to process captured information and represent it in the form of an ontology?
3) How to allow for the graphical querying of provenance and for the traversal of the resulting provenance trees?

We will define the scope of the provenance information to be inclusive of the software dependencies and Operating System (OS) information, as well as relevant inputs and outputs from a process execution. Through the development and assessment of this functional prototype we look to answer the following questions:

1) What constitutes adequate provenance information required for reproducibility?
2) What data is producible from the methodology and approach outlined in section 1.4 in order to obtain the necessary provenance information?

3) How much of this data were we able to reliably capture and represent in a functional manner?

4) What processes lending themselves to reproducibility can benefit from the work presented in this thesis?

5) What gaps in the data and its representation(s) can be identified for further work and research in this subject area?

1.4 Methodological Approach

In order to accomplish the outcome described in section 1.3, we will break the tool up into its individual components. We present here a brief overview of the main components that will make up the solutions.

1.4.1 Versioning

A combination of Git and the NixOS[DLP10] will be used to perform the versioning of both individual files and artifacts as well as the the file-system itself. The information gathered from NixOS and Git will be used to allow for the retrieval of the exact file and environment in which it was produced and/or utilized.

1.4.2 Logging

To strengthen the monitoring abilities detailed in section 1.4.1, we will also need to trace the resources used in activities such as file generation. That is to say, the inputs, outputs, and software dependencies will need to be captured. To do this, we will utilize the Linux Audit Daemon[GRb].

1.4.3 Representation

The information gathered as detailed in sections 1.4.1 and 1.4.2 must now be associated with one another and structured in a well-defined manner. To do this, we will utilize the Prov-O Ontology[LMS13], and provide a solution allowing for the modification of this ontology in an automated fashion utilizing the OWL2 API[HB09].

1.4.4 Interaction

To allow the user(s) to graphically interact with the captured data, we will utilize Stanford's Protégé[Mus15] and its plugins OntoGraf[Fal16] and SPARQL Query [Red16]. Users will be able to query for the desired data by means of the SPARQL[GSP13] query language.

1.5 Thesis Structure

The remaining sections of this thesis are organized as follows:

Chapter 2 provides insight into the topic of reproducibility in the context of computation, as well as an overview of existing approaches.

Chapter 3 dives into the technological concepts and components that have been employed in the prototype of the tool resulting from this work. This chapter also describes a case study providing insight into the resulting output of the solution.

Chapter 4 describes in detail the solution, and how the concepts and components from Chapter 3 work together in conjunction with various resources to achieve the goals outline in section 1.3. This chapter also details the results of the case study described in Chapter 3.

Chapter 5 seeks to provide answers to the questions posed in section 1.3, to reflect on the solution detailed in the previous chapters, and to explore possible points of interest leading from this work,

Finally, Chapter 6 will summarize the work and findings as presented in this thesis.

CHAPTER 2

Related Work

This section aims to familiarize the reader with the current state of affairs in the world of computational reproducibility. It is therefore important that we provide a clear and concise explanation for the concept of "reproducibility" in the context of computational experimentation, and specifically in this paper.

In section 2.1.1 we will explore the distinction between the terms "reproducible" and "replicable", as while at first glance they may seem very similar, there are subtle differences between the two that have largely differentiating implications.

In section 2.1.2 we put the information from section 2.1.1 to work by studying a handful of papers that address the issue of computational reproducibility from a procedural standpoint. That is, papers that seek to change behavior as opposed to creating tools and utilities.

Furthermore, in effort to further motivate the issue and ground it in a real-world example, in section 2.1.3 we will study a case where the issue of computational reproducibility has resulted in undesirable, dangerous, and confusing effects in software used in the health care industry.

And last but not least, we will explore some of the existing approaches that aim to solve this issue. In section 2.2.2 we will discuss perhaps the most brute force approach in which a Virtual Machine (VM) is used to provide later users with the exact environment in which an experiment was performed or application ran. In sections 2.2.1 and 2.2.4 we will examine how workflows have been utilized to provide high-level models of an experiment's sequence of operations, and where the shortcomings of this approach lie.

2.1 Literature Studies

2.1.1 Reproducibility and Replicability

One of the main hallmarks of science and experimentation is the ability to reach the same conclusion regardless of which methods were employed during experimentation. To best understand this, we can turn to the world of traditional science. It is often the case that a multitude of experiments or studies utilizing a wide variety of methodologies and subject groups arrive at, and corroborate, the same conclusion. In fact, according to [SBO$^+$07], "In many areas of science it is only when an experiment has been corroborated independently by another group of researchers that it is generally accepted by the scientific community." [Dru09] argues that this would not be the case if an experiment were simply replicated.

It is at this point that the distinction between "replicating" and "reproducing" becomes necessary. Consider two experiments that both reach the conclusion that the earth is round. Experiment A reaches this conclusion by observing the shadows of objects at different locations (in fact, this is how the Greek astronomer Eratosthenes reached this conclusion). Experiment B reaches this conclusion by using a telescope to observe the varying star constellations in the sky (an observation made by Aristotle). While it is of course common knowledge that the earth is round, we can draw some important conclusions from these two experiments. First, we observe that in order to confirm that the earth is round, it is not the case that one can only do so by performing the exact same procedure as a previous experimenter. More specifically, Aristotle's conclusions are not invalid because they were not reached by replicating Eratosthenes' experiment. In the same light, Eratosthenes' conclusions are not invalid because they were not reached by replicating Aristotle's experiment. In fact, [Dru09] goes on to say, "that the greater the difference from the first experiment, the greater the power of the second". The second, and perhaps the more powerful observation, is that it is not the experiment itself that is accepted by the scientific community, but rather "the idea that the experimental result empirically justifies" [Dru09].

Drawing still from the previous example, we can now see where the difference between "reproducible" and "replicable" lies. Would Aristotle or Eratosthenes simply have (re-)performed the same experiment as the another, using the same devices and techniques, the experiment would have simply been replicated. But we must now ask ourselves, what does replicating an experiment allow us to conclude? One should be able to clearly see that replicating alone does not allow one to scientifically reason that the experiment's conclusions are valid and true, but rather provides only the ability to confirm that the conclusion is a result of experiments process' and methodologies. It is only "reproducibility" in its most traditional form that allows experimenters to agree on the common idea that is reached by a multitude of various experiments

Moving forward, we would like to take these observations and "map" them back to the world of computational experimentation. It is argued in [SBO$^+$07] that "Reproducibility would be quite easy to achieve in machine learning simply by sharing the full code used for experiments". We can see here that simply re-using the same code would be akin to the to Aristotle or Eratosthenes using the same tools and instruments from the others experiment, resulting in merely replicating the others experiment. However, we can also make two additional conclusions from this argument:

1) We should be careful to not disregard the idea of "replicability" as something not worthwhile or valuable.

2) The definition of "reproducible" in the context of computation experimentation is often still a blurry one lying somewhere between "replicable" and "reproducible" as we have previously defined them.

In fact, there are many situations where it is the replication of the exact conclusions that is required in order to trust an experiments veracity, as we will see in section 2.1.3. Additionally, this is often the case when users wish to build upon the conclusions of a previous experiment, and being able to reliably replicate these results is critical in doing so.

Finally, we now clarify where in this blurry spectrum the definition of "reproducible" as used in the context of this thesis lies. We choose a definition that is more in line with the idea of "replicable" as defined in this section. That is to say, the prototype tool tool resulting from this thesis aids itself more to the act of recreating the exact same environment, resources, and process' used in the execution of the original experiment and/or application.

2.1.2 Reproducibility Research

This section provides the reader with a brief overview of a variety of research papers published within the context of computational reproducibility. These papers revolve around the theme of analyzing the current climate of the field and suggest procedures and rules to help researchers make it one more susceptible to reproducibility.

It perhaps makes the most sense to start by addressing the culture of the computational research scientific community. [Pen11] offers insight into this culture by arguing that the "biggest barrier to reproducible research is the lack of a deeply ingrained culture that simply requires reproducibility for all scientific claims", and that a "culture of reproducibility" must be developed. This barrier is further explored in [CPM$^+$14], where the authors attempted to reproduce the results from a total of 613 published papers. [CPM$^+$14] reached out to the various publishers asking for source code for the published results or assistance in reproducing the experiments when necessary. The results of their efforts can be Figure 2.1. In total, 260 emails were sent, of which 30 (11.5%) received no response, 81 (31.1%) received a "yes" response in which the publishers provided their source code, and 149 (57%) received a "no" response in which the publishers refused to provide their source code. These anecdotal results seem to indicate the absence of a

Group	#	Practical	Emails	Email Reply			Build		Run		Reproduci-
			not sent	None	“yes”	“no”	fails	OK	fails	OK	bility
ASPLOS’12	37	30	7	1	4	15	3	4	0	4	17.4%
CCS’12	76	56	18	2	15	11	12	13	4	9	23.7%
OOPSLA’12	81	66	5	1	9	15	22	23	2	21	34.4%
OSDI’12	24	24	7	0	2	7	4	6	1	5	29.4%
PLDI’12	48	48	7	1	10	14	17	9	5	4	9.8%
SIGMOD’12	46	45	20	1	8	7	6	11	2	9	36.0%
SOSP’11	28	26	7	1	3	9	7	2	0	2	10.5%
TACO’9	60	39	2	5	5	17	8	7	0	7	18.9%
TISSEC’15	13	6	0	0	1	2	2	2	0	2	33.3%
TOCS’30	14	13	0	2	0	6	3	2	0	2	15.4%
TODS’37	29	17	0	5	3	3	3	6	0	6	35.3%
TOPLAS’34	16	11	2	0	1	0	5	4	0	4	44.4%
VLDB’12	141	134	30	11	20	43	16	34	7	27	26.0%
NSF	255	227	54	12	37	58	50	53	9	44	25.4%
No NSF	358	288	51	18	44	91	58	70	12	58	24.5%
Academic	415	350	63	20	62	86	85	96	12	84	29.3%
Joint	149	130	36	7	17	45	19	23	8	15	16.0%
Industrial	49	35	6	3	2	18	4	4	1	3	10.3%
Conferences	481	429	101	18	71	121	87	102	21	81	24.7%
Journals	132	86	4	12	10	28	21	21	0	21	25.6%
Total	613	515	105	30	81	149	108	123	21	102	24.9%

Table 2.1: Summary of results from [CPM$^+$14]; Source: [CPM$^+$14]

"culture of reproducibility” as discussed in [Pen11], and instead indicate a reluctance to help and a culture of proprietary algorithms.

In an attempt to directly address this issue, [SNTH13] presents ten simple rules to guide researchers towards more useful, and reproducible computational researcher. We address here only those that relate most closely to the work in this thesis.

Rule 1: For Every Result, Keep Track of How It Was Produced
: This may seem like an intuitive rule, but it is also one that is easily forgotten or inefficiently followed. [SNTH13] suggests that as opposed to manually documenting an experiment steps, researchers should utilize automated tools, such as Workflows (see sections 2.2.1 and 2.2.4).

Rule 3: Archive the Exact Versions of All External Programs Used
: A common problem with many of the current reproducibility solutions, including workflows, is the ability to obtain a 100% identical working environment. We will later see (section 3.1.2) how traditional package managers can be adapted to help solve this problem.

Rule 4: Version Control All Custom Scripts
: In large toolchains, small changes can result in repercussions that are sometimes

very difficult to track down. By versioning all used utilities and scripts, one can easily rewind and fast-forward changes to help narrow down the cause. We will see in sections 3.1.4 and 4.3.5 how we this idea is utilized in the thesis' work.

Rule 10: Provide Public Access to Scripts, Runs, and Results
This rule speaks mostly to the work performed in [CPM+14]. The computational research communty must adapt the conventional scientific community's practice(s) of being open, honest, and clear with ones results and experimental methods. It will only be then that work in this field will be more easily corroborated and trusted in its veracity.

[SNTH13] acknowledges that while their presented rules may sometimes not be strictly adhered to in light of deadlines and other factors, that at the bare minimum "you should at least be able to reproduce the results yourself". Once having achieved this, [SNTH13] proposes two extensions for future work:

1) To go from a level where you can reproduce results in case of a critical situation, to a level where you can practically and routinely reuse your previous work and increase your productivity.

2) To ensure that peers have a practical possibility of reproducing your results, which can lead to increased trust in, interest for, and citations of your work.

2.1.3 FreeSurfer

We now turn our attention to the FreeSurfer application. FreeSurfer is a free and open source software suite developed by the Laboratory for Computational Neuroimaging at the Athinoula A. Martinos Center for Biomedical Imaging. The FreeSurfer suite allows users to process and analyze human brain Magnetic Resonance Imaging (MRI) images. This application suite is of interest as a motivating example for the work in this thesis in that issues surrounding its use coincide nicely with our working definition of "reproducible" as defined in section 2.1.1. Specifically, [GHJ+12] uncovers issues surrounding the accuracy of the FreeSurfer application in its measurements of cortical thickness and volumes of neuroanatomical structures. Additionally, [GHJ+12] details how the discrepancies in measurements stem from variables such as the FreeSurfer version, the OS the application is ran on, and the hardware platform. A direct consequence of these discrepancies is the inability accurately detect changes between measurements performed across multiple systems.

We see in Table 2.2 the various hardware platforms and software versions that were used by [GHJ+12] in the analysis of FreeSurfer. The takeaway from this table is that a multitude of variables that can effect the results of the FreeSurfer application. This naturally leads to questions such as: Does the number of CPUs affect the result(s)? How do the different software dependencies such as libraries and the OS itself affect

Name	Type	OS	CPU	N[a]	RAM
iMac1	iMac	OS X 10.5.8	3.06 GHz Intel Core Duo	2	8 GB 1067 MHz DDR3
iMac2	iMac	OS X 10.6.5	2.8 GHz Core i7	8	16 GB 1067 MHz DDR3
MacPro1	MacPro	OS X 10.5.8/10.6.5[b]	2×3.2 GHz Quad-Core Intel Xeon	8	16 GB 800 MHz DDR2
MacPro2	MacPro	OS X 10.6.4	2×3.0 GHz Quad-Core Intel Xeon	8	16 GB 1066 MHz DDR3
MacPro3	MacPro	OS X 10.6.4	2×2.6 GHz Quad-Core Intel Xeon	8	16 GB 1066 MHz DDR3
HP	HP	CentOS 5.3	2×2.66 GHz Quad-Core Intel Xeon	8	16 GB 667 MHz DDR2

[a]N is the number of processors.
[b]By means of an external disk this workstation could run under two different OS versions.
Note: All Macintosh workstations used the UNIX shell and the Hewlett-Packard (HP) workstation the LINUX shell. OSX 10.6.4/10.6.5 was used in 32 bits mode, whereas CentOS was used in 64 bits mode.
doi:10.1371/journal.pone.0038234.t001

Table 2.2: Workstations used in the analysis of FreeSurfer; Source: [GHJ+12]

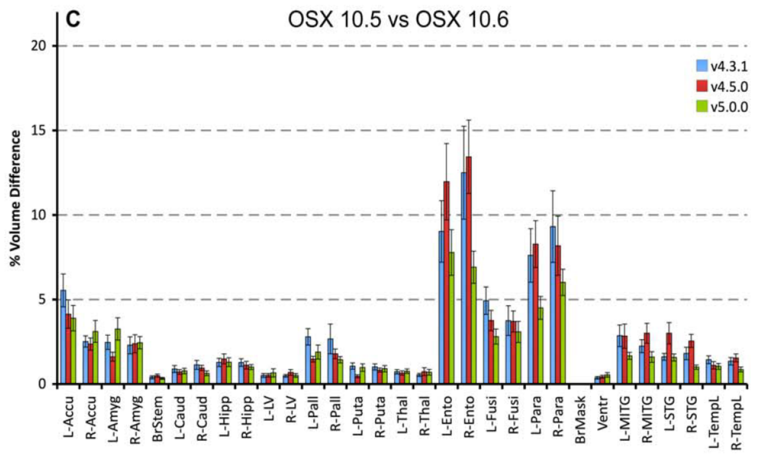


Figure 2.1: Effects of data processing conditions on the voxel volumes for a subsample of (sub)cortical structures; Source: [GHJ+12]

the measurements? What impacts does the hardware itself have on the FreeSurfer application? These are questions that are core to the issue of reproducibility. To further motivate the impact of these variables, Figure 2.1 depicts the percentage of volume differences in the measurements of (sub)cortical structures between analysis performed on Mac OSX 10.5 and Mac OSX 10.6 with different FreeSurfer verions. We observe right away the measurement discrepancies between OS and FreeSurfer versions.

We note here that this is a situation in which a solution allowing for the exact replication of the original execution is necessary. [GHJ+12] concludes that FreeSurfer users should "provide not only the version of FreeSurfer that was used, but also details on the OS version and workstation". This is precisely the information that the work in this thesis aims to provide to the user as described in section 1.3, detailed in Chapter 4, and captured in our working definition of "reproducible".

2.2 Existing Approaches

This section provides the reader with an overview of the existing approaches attempting to solve the problem of computational reproducibility. We will explore some of the leading and cutting edge solutions in regard to how each approaches the issue of reproducibility, the central components that make up the solution, and their pitfalls and shortcomings.

2.2.1 Workflows

As eluded to in the introduction, computers find themselves playing increasingly key roles in the scientific community. A direct result of this has been an explosion in the amount of data produced and made available to researchers. However, due to the lack of scientific standards enforcing the use and development of this data, we find that the data is spread across heterogeneous Database Management Systems (DBS) and data structures [Rom08].

To combat this, researchers developed a form of experimentation called *in silico* experimentation. *In silico* (Latin for "in silicon") experimentation allows for researchers to perform research experiments utilizing computational models and tools while providing access to information repositories [OAF+04]. However, *in silico* experimentation presents a new issue: How can one represent the steps and procedures of an experiment in such a way that allows for reproducibility at a future time? Workflows attempt to solve this problem by allowing researchers to define a "precise description of a scientific procedure — a multi-step process to coordinate multiple tasks, acting like a sophisticated script" [HTT09].

The benefit of utilizing workflows is two-fold: (1) it allows researchers who are not experts in fields such as programming or web services to not only have access to data, but to be able to (re-)execute experiments themselves [HWS+06], and (2) "reduces the overheads of downloading, installing, and maintaining resources locally whilst ensuring access to the latest versions of data and tools" [WFDRG09].

To manage these workflows, Workflow Management Systems (WFMS) have been developed to assist reseachers by providing a number of benefits and ease-of-use features. WFMSs implement environments in which the execution of workflows can be (1)invoked, (2) monitored and recovered from failures, (3), optimized for memory, storage, and execution, (4) handled properly in terms of data management, (5) logged, and (6) secured and monitored with regard to access polices [HTT09]. In Table 2.3 we find a summary of three popular WFMSs; Taverna [MSRO+10], Kepler [LAB+06], and Activti [Rad12].

However, workflows and WFMSs do not always allow researchers to successfully reproduce previous work. [MR15] examines a total of 1443 workflows and provides statistics with regard to what percentage of the workflows were re-executable, and what issues were faced in attempting to do so. We see in Table 2.4, that out of the 1443 examined workflows, only 917 (64.55%) were runnable (able to be open/ran by a WFMS).

Engine	**Implement.**	**Script Language Support**	**Designer Support**	**Execution Engine**	**Provenance**
Taverna	Java	Beanshell	Standalone	Integrated with designer	Database (Apache Derby)
Kepler	Java	Python	Standalone	Integrated with designer	Database (HSQLDB)
Activiti	Java	JavaScript, Python, Ruby, etc.	Eclipse IDE	Web application or Java program	Database (H2 DB)

Table 2.3: Features of workflow systems; Source: [MRM16]

Table 2.5 indicates that only 731 (50.7%) of the 1443 workflows were executable, and Table 2.6 shows us the execution results of these workflows, with only 341 having executed successfully, or "29.2% of the original data set of 1,443 objects" [MR15]. The authors go on to state that the majority of the workflows that are not re-executable suffered only from trivial issues, but we leave this to the reader to further explore.

Total workflows	1443	
Workflows with no input ports	345	23.91%
Workflows with input ports	1098	76.09%
Workflows with no example values	429	29.73%
Workflows with some example values	97	6.72%
Workflows with all example values	572	39.64%
Workflows that can be run	917	63.55%

Table 2.4: Workflow Input Port Statistics; Source: [MR15]

Initial data set	1443
Removed – missing input values	526
Removed – disabled processors	180
Removed – not executable in test environment	6
Executable workflows	731

Table 2.5: Workflow Executable Data Set; Source: [MR15]

Execution successful	341
Execution failed	364
REST service not reachable	4
REST service not authenticated	4
Other missing authentication (WSDL, RShell, ...)	30
Resource/File not available (local or remote)	13
Tool command not available	15

Table 2.6: Workflow Execution Results; Source: [MR15]

2.2.2 Virtual Machines

VMs offer a brute-force and arguably simple approach to solving the problem of computational reproducibility. It allows researchers to deliver what [HBC15] refers to as a "virtual reference environment" with a publication to facilitate the reproduction of the publications results. Simply put, a VM affords users the ability to work on identical software environments with regard to OS, applications, libraries, and configuration.

In fact, using a VM allows researchers to adhere to many of the rules presented in [SNTH13] and outlined in section 2.1.2. Specifically, Rule 3 is obeyed by providing a VM that comes with the exact version of all used programs, and rule 10 is satisfied by packaging all the utilized scripts, runs, and results into the VM. [HBC15] goes on to list the potential obstacles one might encounter in the distribution of VMs with published papers.

The first and perhaps most obvious issue is that of size. VMs can very quickly requires gigabytes of storage space, making the downloading of said VM not a realistic requirement for future work. Other issues include things like "Issues of curation", "Licensing and distribution", and "Specific architecture requirements" [HBC15], which we leave to the reader to explore in more detail.

Furthermore, if one is looking for a solution that affords reproducibility in its most traditional form (see section 2.1.1), then it should be quite clear that VMs are not equipped to handle the task. VMs do little more than allow users to replicate what once was. They do not intrinsically encode a process' steps, govern ones execution, provide information such as provenance, or allow the pipelining of future work. Through the work of this paper, we aim to provide these missing features, some of which are necessary even for the task of reproducibility as defined for use in this work (see section 2.1.1).

2.2.3 Docker

Docker is a relatively new (released in 2013) open source utility that offers a new twist on some tried and true concepts. Specifically, Docker combines the approaches of Linux Containers (LXC), OS virtualization, and version control utilities [Boe15]. Although Docker is marketed more towards the business sector, it has gained much popularity in the fields of informatics and computational research [PF16].

At its core, Docker implements user-friendly access to LXCs, allowing users to build, execute, and share them [PF16]. While LXCs, still require an underlying Linux OS, they allow applications and their runtime environments to be isolated from the underlying OS. A Docker Container (an LXC with additions from Docker) is defined through what Docker has aptly named a *Dockerfile*. A Dockerfile is a human readable text based "recipe" that defines the "necessary software dependencies, environment variables and so forth needed to execute the code" [Boe15]. The Dockerfile can then be easily saved, versioned, shared, and later used to build the original Docker Container when desired. Additionally, once built, the resulting container can be exported as a binary file for ease of sharing, and are usually smaller than a VM [PF16].

We see in Figure 2.2 how (Docker) containers are layered with the systems software and hardware components. We reiterate here a few important distinctions between a VM and Docker Container. (1) Docker Containers do not contain any OS components. They only serve to isolate applications and their dependencies from the underlying (OS) software. (2) Subsequently, Docker Containers are much smaller in size than VMs, and thus more easily shared. (3) VMs behave much more like a black box, while Docker Containers provide clear insight into their contents and how they were created.

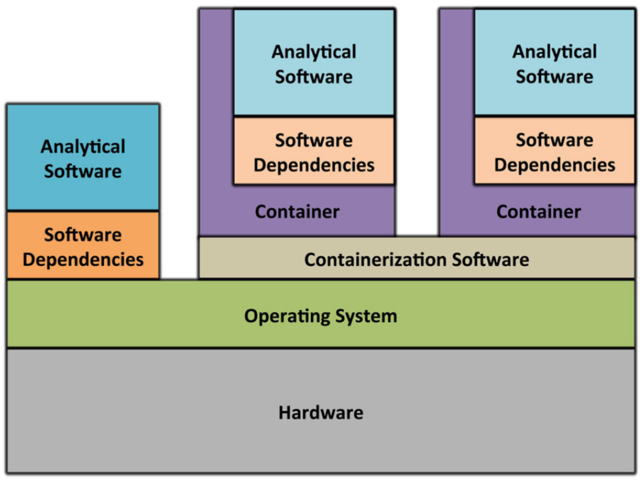


Figure 2.2: Container Layer Example; Source: [PF16]

However in term of true reproducibility, Docker suffers from some serious flaws, one of which lies in the Dockerfile. Dockerfiles rely on the package manager of the underlying Linux OS to obtain the dependencies required to build the Docker image. This means that images built at different times "will install more recent versions of the same software" and as a result will not always be bitwise identical [Boe15]. While it may be possible to configure the package managers to avoid this situation, this is outside the feature-set of Docker itself. Thus, Docker fails to provide a reliable solution for reproducibility both in its traditional sense, and how it is defined for the work in this paper.

2.2.4 Vframework

In attempt to solve some of the issues with traditional workflows and WFMS, [MPS+13] proposes Vframework; a framework for the verification of preserved processes. In section 2.2.1 we saw that for a variety of reasons, only 29.2% of the tested workflows were re-executable. Vframework attempts to mitigate these issues by performing complex analysis techniques on the original workflow environment, as well as verification and validation on the redeployment environment. In fact, Vframework does not only serve to ensure that a workflow is re-executable, but also that the results of the workflow are identical at the time of re-execution.

Vframework accomplishes these tasks through seven steps that describe what information is required to successfully verify a redeployed process, and how to capture that data. Figure 2.3 depicts these steps of whose important details we will quickly cover here.

Step 1: Describe the Original Environment
: It is necessary to define a process' context by identifying its environmental dependencies. Additionally, potential redeployment scenarios must be considered to allow for the identifying of relevant and significant properties to be saved and utilized in the redeployment process.

Step 2: Prepare the System for Preservation
: The interactions of the process must be identified. This includes a process' inputs, outputs, configurations, and external influences. Together, this information works to ensure a deterministic execution of the process at the time of redeployment.

Step 3: Design Verification Settings
: In order to accurately verify a process' execution at the time of redeployment, measurement points in the execution must be identified. These measurements work to ensure that the correctness of the process execution.

Step 4: Capture Verification Data
: This step involves the configuring of the tools and environment to be used in the process of capturing the verification data. Additionally, the captured data itself is verified for its correctness.

Step 5: Prepare System for Redeployment
: The system being used for redeployment must be (1) configured for the capturing of performance data, (2) redeploy the process on the new system, and (3) execute the process.

Step 6: Capture Redeployment Performance Data
: Performance data must be captured and verified. Additionally, the performance data will be assessed to verify if the process was executed in a deterministic manner.

Step 7: Compare and Assess
: Finally, the original and redeployed execution results will be compared with one another and a summary report detailing the results created.

We can see that Vframework offers a very comprehensive solution for the reproducibility of computational research, and is by no means a flawed approach. However,

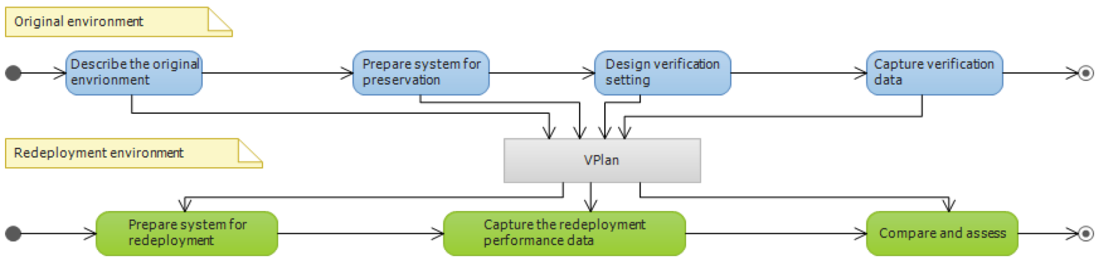

Original environment
Describe the original envrionment
Prepare system for preservation
Design verification setting
Capture verification data
VPlan
Redeployment environment
Prepare system for redeployment
Capture the redeployment performance data
Compare and assess

Figure 2.3: Vframework Outline; Source: [MPS+13]

as the core ideology of Vframework revolves around the capturing and redeploying of workflows, many of the same issues we have outlined in section 2.2.1 will also be faced here.

2.3 Summary

In Chapter 2 we have covered a number of key topics critical to the remainder of this paper by reviewing relevant literate and exploring a handful of existing approaches. Most importantly, we have defined the term *reproducibility* within the scope of the work presented: exact replication. We then uncovered some of the philosophical and cultural issues that surround reproducible research within the computational research community, and discovered that for reproducibility to gain traction within the community, a cultural shift must occur. In [CPM+14], the authors arrived at this conclusion after encountering much resistance from the publishers of other papers when asked about their work. To observe the obstacles that analysis procedures face when executed across multiple software versions and hardware platforms, we studied the Freesurfer application as presented in [GHJ+12] and observed a number of discrepancies in the results of the applications.

Moving on to existing approaches, we covered two general techniques for dealing with reproducibility: workflows and VMs. We discovered how workflows can make complex systems more accessible for researchers who aren't experts in programming, and also how they can quickly degrade and become unusable [MR15]. It should also have been clear from section 2.2.2 that VMs do not provide an adequate solution moving forward. We also reviewed two existing tools/utilities: Docker and VFramework. We found that while both offervery comprehensive and useful features, each leaves more to be desired. Docker does not provide a solution that allows users to truly replicate the original environment, and Vframework, relying on workflows, will ultimately fall victim to many of the same issues as detailed in section 2.2.1.

CHAPTER 3

Methodology

This chapter aims to inform the reader of the methodology employed in the development of the prototype tool presented in this paper. In doing so, we must familiarize the reader with a number of concepts that are incorporated into the various components of the final solution. In each subsection of section 3.1, a new concept will be introduced and explained with the aid of examples and use-cases. For the unfamiliar readers, this section will prove crucial in the continued reading of this thesis, specifically in Chapter 4 where the concepts defined in this chapter will be implemented and adapted to the presented work.

In section 3.1.1 we will review the Semantic Web and how ontologies, specifically the OWL2 Ontology Language [MP12], build upon the Resource Description Framework (RDF) by applying semantics to the structure it provides. We will continue to explore this topic studying how the Prov-O data model utilizes the OWL2 Ontology Language to express provenance.

Futhermore, in section 3.1.2 we will explore how package managers are used to automate the process of installing and configuring software on an OS. We will also briefly cover some of the issues that traditional package managers exhibit with regard to reproducibility as motivation for the use of the Nix functional package manager in the presented work.

In section 3.1.3 we will explore what "functional" means within the scope of programming. Additionally, we will study NixOS [DLP10] and how it applies this concept to create a OS that aspires to implement a purely function model for not only package management, but also system configuration. In our study of NixOS, we will also review the previous sections to explore how they are implemented within NixOS itself.

The concept of version control software will be explained in section 3.1.4. This component is crucial to the proposed solution, as readers will discover in Chapter 4.

In section 3.1.5 we will present the Linux Audit Daemon [GRb]. This component of the solution is of high importance as it allows for the collecting of data to be parsed and utilized in the remainder of the protoype tool presented in this paper.

We must then describe methodologies in which the user can view and interact with provenance information. In sections 3.2.1 and 3.2.2 we will introduce how we can use OntoGraf [Fal16] and SPARQL [GSP13] to accomplish this task.

Finally, in section 3.3 we will illustrate two working examples which we will later use to present the results compiled from the combined work of the concepts and components in the previous section.

3.1 Concepts and Components

3.1.1 The Semantic Web

The World Wide Web (WWW) is a seemingly endless resource of information, allowing anyone to access data for almost any topic imaginable. In 2001, Tim Berners-Lee, the "founder" of the WWW, realized that as the WWW grew increasingly larger and with it the amount of data, that it lacked structure beyond web-page layouts and links. To bring structure to the content of the WWW, Berners-Lee introduced concept of The Semantic Web [BLHL01]. It is in [BLHL01] that we can find perhaps the most concise definition of The Semantic Web; "The Semantic Web is not a separate Web, but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation". It is important to note here why this definition, or concept, is truly significant. It may be easy to overlook, but the WWW was not designed for computers to interpret and manipulate in a meaningful manner, but rather for humans to read. Without structure, any sort of automated reasoning about a web-page and its content would not be possible. In the remainder of this section, we explore the structure that The Semantic Web has introduced, and how it applies to the work in this thesis.

Before doing so, let us first bring the concept of The Semantic Web into the context of this thesis. While it was originally conceived to bring structure to the WWW, the very same concepts that make up The Semantic Web can be applied to data in general. Just as the WWW saw (and continues to see) an explosion in data, so does the scientific community. Furthermore, this data is not homogeneous data, and thus questions such as "How do I use this data?", "What purpose does this data type serve?" and "What does this data mean?" will be asked and must be answerable.[HTT09]. Ideally, these questions should be answered through the use of tools that are able to infer the meaning of data, and the relationships that exists between the various data points.

We must now define a few key terms that are crucial in understand the layers of The Semantic Web. As seen in Figure 3.1, the *Uniform Resource Identifier (URI)* is the most fundamental layer. The URI gives an object a unique and unambiguous name that serves to help identify where that object came from [Bra07]. As an example, consider the situation in which we want to identify a person named Alice. We can indicate where someone can find Alice in the phone book with the following URI: **http://www.examplephonebook.com/people#Alice**. In this particular example, we have chosen to utilize the hash (#) notation, allowing us to specify that the indicated resource is not a web-page or document [CWL14]. Thus, the prefix of the URI for Alice is then simply **http://www.examplephonebook.com/people**

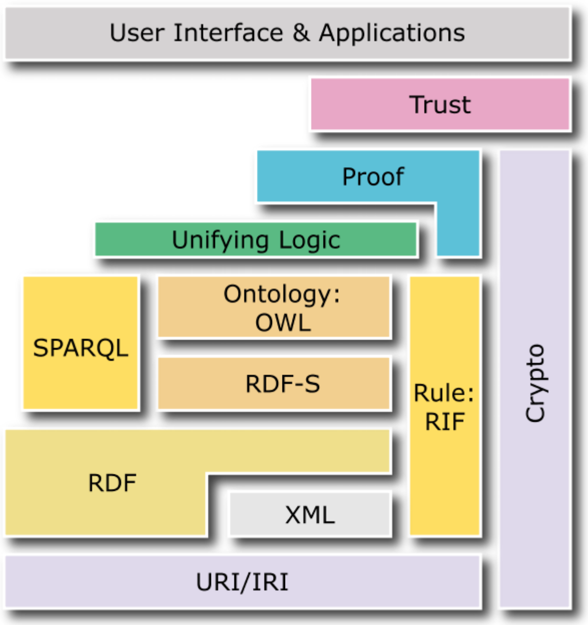


Figure 3.1: Layers of The Semantic Web; Source: [Bra07]

We focus now on the RDF layer. This layer allows data to be linked together by what is called a *triple*. A triple consists of three components as seen in Figure 3.2; the subject, the predicate, and the object. We can then use triples to create relationships between objects represented by URIs, giving them syntactical meaning. We note here that a set of triples is called an RDF Graph. Consider the following example in which we wish to indicate that Alice is the daughter of Bob. Here, *Alice* is the subject, *isDaughterOf* the predicate, and *Bob* the object. In this example, the object, like the subject, is an URI. However, it may also be the case that the object be a literal, with the predicate indicating that the object has the specific literal value.

<http://www.examplephonebook.com/people#Alice>
<http://www.relationships.com/isDaughterOf>
<http://www.examplephonebook.com/people#Bob>

The last important and relevant layer to this work is the ontology layer. The full expressive power of ontologies will be left to the reader to discover, and we cover here

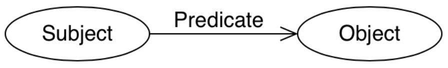


Figure 3.2: An RDF Triple; Source: [CWL14]

the extensions from the RDF layer both most relevant to the context of this work, and to the fundamental understanding of ontologies and the OWL2 Ontology Language.

The ontology aims to solve many issues that can arise when using only RDF to structure data. Consider the scenario where two different databases utilize two different URIs for what is in fact, the same concept. The ability to discern these two concepts as one and the same is enabled by the ontology layer through the expression of taxonomical structures and inference rules. Taxonomy allows one to define classes and the relationships among them, including explicit equivalence relations between URIs.

This entails a large number of benefits and increased expressive power through the use of class properties and subclass inheritance rules. As an example of class properties, we can specify that "an address may be defined as a type of location, and city codes may be defined to apply only to locations" [BLHL01]. Such a specification places restrictions on classes and their capabilities, and the ability for these specifications to be inherited by subclasses allows for a large number of relations to be expressed.

Taking the next step, we can use these properties to infer further information from triples. For example, "If a city code is associated with a state code, and an address uses that city code, then that address has the associated state code" [BLHL01]. The inference rules in the ontology layer thus allow computers to deduce new information and to manipulate it according to predetermined rules in a way that is meaningful to the user [Kuc04][BLHL01].

Now that we understand the ontology layer, let us briefly examine the OWL2 Ontology Language. Ontologies, as seen in the previous paragraphs, in essence provide the means to establish classes and semantic relationships between them. How those relationships are defined, how they are organized, and to what extent information can be deduced and manipulated, is dependent on the ontology, or vocabulary, they are implemented in. In the case of the OWL2 Ontology Language, there is a very high degree of expressive power. Classes and objects can be described by relationships such as unions, intersections, and disjointness, properties can be constrained by cardinality restrictions, and also extended by functional relationships such as transitivity, inverses, and symmetry [MP12]. An overview of the OWL2 structure can be seen in Figure 3.3.

The presented work utilizes the PROV Ontology (PROV-O); an ontology that "expresses the PROV Data Model using the OWL2 Web Ontology Language" [LMS13]. Though not the focus of the section, we note that the PROV Data Model enables the representation of provenance information in a model that is interchangeable between systems [MM13].

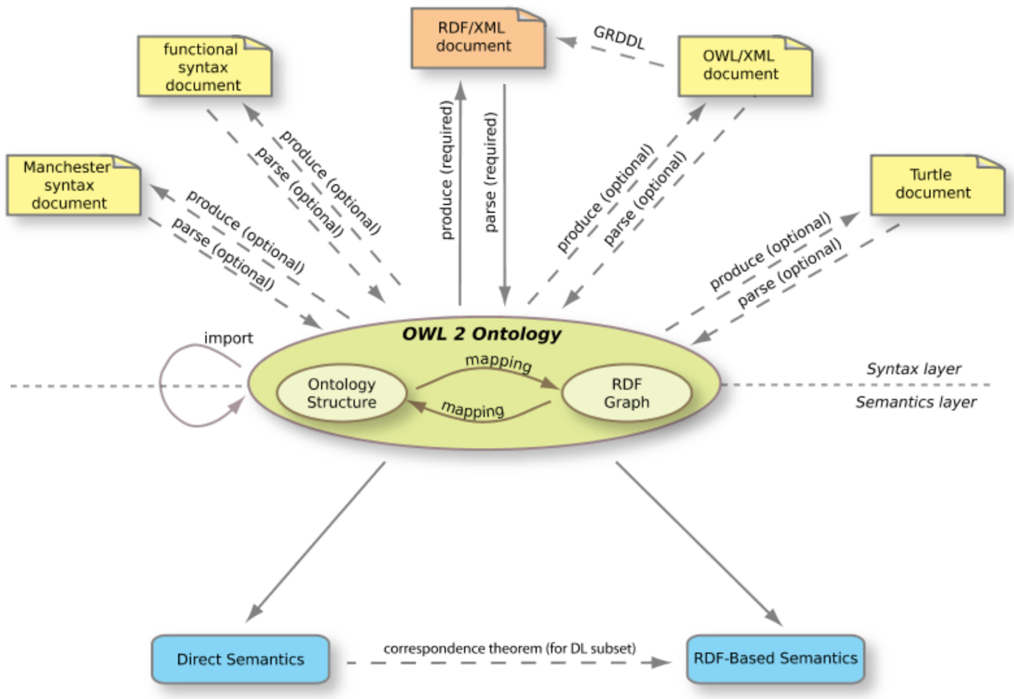


Figure 3.3: OWL2 Ontology Structure; Source: [MP12]

PROV-O Internationalized Resource Identifiers (IRI) (URIs extended to the Unicode typeset) fall into three main categories: Starting Point Terms, Expanded Terms, and Qualified Terms [LMS13].

Starting Point Terms
: Starting Point Terms encompass the classes and properties that allow for the expression of simple provenance information. The three classes in PROV-O are *prov:Entity*, *prov:Activity*, and *prov:Agent*, each of which can be related to the others by the use of the property terms. Some of the available properties include *prov:startedAtTime*, *prov:wasGeneratedBy*, and *prov:wasAssociatedWith*.

Expanded Terms
: Expanded Terms provide the ability to indicate additional information in the provenance of the three PROV-O classes mentioned above. Examples include classes such as *prov:Collection*, and properties such as *prov:hadMember* and *prov:value*

Qualified Terms
: Qualified Terms allow for even more detailed provenance information, in that they enable the specification of attributes for relations existing between terms in the previous two categories. For example, *qualifiedGeneration* can be used to provide

more details about the generating entity including identification information, attributes, and comments.

We will further explore the the PROV-O IRIs as necessary in section 4.3.7. Below in Figure 3.4 we see an example of how the above definitions can relate with one another.

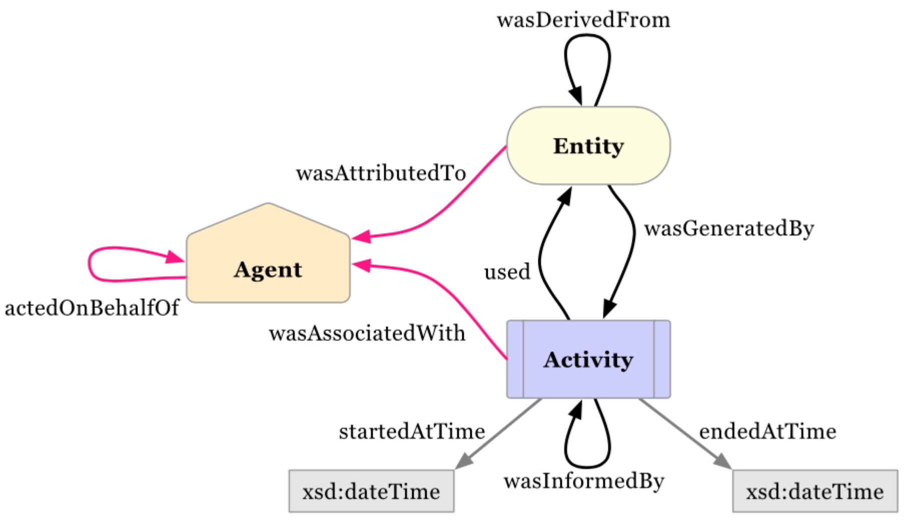


Figure 3.4: PROV-O Example; Source: [LMS13]

3.1.2 Package Managers

Methods in which software can be reliably employed and delivered has long been a point of contention. Dating back to early versions of Microsoft Windows, the installation of various programs often led to what has been dubbed Dynamic Linked Library (DLL) hell; a situation in which the installation of one program and its dependencies breaks the functionality of another. Package managers, through the use of packages, provide an easy and automated solution to DLL hell and the process of obtaining and upgrading software. Packages provide the means in which all required software source code, dependencies, licenses, documentation, and installation specifications can be combined into a single deliverable. [Spi12].

Package Managers, in their traditional sense, operate within an imperative model. This entails that the actions performed in the updating of a package are done in a stateful manner, resulting in the active destruction of files as they are updated. The implications of this are sometimes subtle, but always far reaching. On Unix systems, files relevant to installed packaged are conventionally installed to Unix's hierarchical file system. That is to say, dependencies and source code more often than not end up finding themselves under directories such as */bin*. The issue stems from the observation that objects in such directories offer no transparency to software requiring its invocation. Put another way, software invokes dependencies through the use of pointers to global mutable variables, without assurances as to what version of the dependency, and subsequently its behavior, is being invoked. [DLP10].

We now briefly explore some shortcomings of imperative package managers as well as motivations for moving towards functional package managers as described in [DdJV04].

Variability
: Flexible systems must support the presence of multiple variants of a dependency on the same system. Software should not be broken when multiple variants of the same dependency with different optional features exist on the same system.

Consistency
: Similar to variability, multiple versions of a dependency on the same system should not break installed software. The installation of new versions should not be destructive in the sense that it deletes or overwrites files from the previous version, on which current software may depend.

Atomicity
: Upgrades are not atomic. This means that during an upgrade process, software reliant on the affected dependencies may not function correctly. This affects more than just individual components, as it is often the case that system-wide shared libraries must be updated before others.

Identification
: Software often specifies its dependencies by version number alone. Lacking is the configuration specifications and build parameters that were used at the time of compilation. Thus, version numbers by themselves cannot act as unique identifiers for software packages.

Source/Binary Development
: Source code and its corresponding binary package do not necessarily have a 1-to-1 correspondence. That is to say, the compiling of source code often requires explicit actions from the user with respect to desired features and options, resulting in binary images modeling a particular variant of the source code.

Centralized vs Local Package Management
: Package deployment to a network of computers should ideally not require the individual installation on each individual computer. Package Managers should handle software deployment centrally, and allow individual users to adapt the installations to their specific requirement should need be.

The Nix package manager has been specifically designed to address these issues, and does so by utilizing a purely functional model. Nix thus delivers immutable packages solely dependent on the inputs to the functions from which they were built. There are three main concepts that make up Nix: (1) Nix expressions, (2) the Nix Store, and (3) generic means for sharing build results. The remainder of this section will explore these three concepts, and their application to the presented work will be explored in section 4.3.1 [DdJV04][DLP10].

Nix Expressions

A Nix expression, detailed in Figure 3.5, is a function that takes a set of arguments as seen at point [1]. These arguments are the dependencies required to build the xmonad package. The output of the function is a derivation, described at point [2]. The derivation contains an attribute set of the form *{name = value;...}*. This attribute set contains the most important information for building a package: the name, the fetchurl for the source, the build inputs (point [3]), the configuration (point [4]), build instructions, and installation instructions. All name values, with exception of the meta data at point [7], are subsequently passed as environmental variables to the build script provided by the *stdenv* dependency [DdJV04][DLP10].

Also of significance is the *$out* variable at point [6], indicating the output path of the package resulting from the build of the derivation. As a Nix expression is a function, it must (1) be called with valid arguments, and (2) can be called any number of times, always resulting in a new instance of the built package. The most important implication of these two points is that each created instance is independent from the other, and, unlike in the case of imperative package management, do not interfere with each other. This is ensured by the *$out* variable, whose importance is further explained in the description of the Nix Store below [DdJV04][DLP10].

The Nix Store

In Figure 3.6 we see an example of the Nix Store directory structure. We briefly explain the directory hierarchy, and how it maintains a purely function approach.

Where in this structure a built package finds itself is dictated by the *$out* variable at Figure 3.5 point [6]. It must be ensured that a package's location does not interfere with other variants of the same package. This is accomplished by utilizing a 160-bit hash of the derivation resulting from the evaluation of a Nix expression, and more specifically, the attribute set of the resulting derivation. We reiterate here that a package's derivation contains not only its inputs and dependencies, but also its configuration. The resulting hash is then used to create the directory name where the built package is to be stored. This is demonstrated in the following example from [DLP10]:

/nix/store/8dpf3wcgkv1ixghzjhljj9xbcd9k6z9r-xmonad-0.5/

In Figure 3.6, solid arcs are used to denote references to dependencies. These dependencies include inputs to the Nix expression for the built package. Furthermore, the Nix Store must ensure atomic operations, which is accomplished

```
{ stdenv, fetchurl, ghc, X11, xmessage }: [1]

stdenv.mkDerivation [2] (rec {
  name = "xmonad-0.5";

  src = fetchurl {
    url = "http://hackage.haskell.org/.../${name}.tar.gz";
    sha256 = "1i74az7w7nbirw6n6lcm44vf05hjq1yyhnsssc779yh0n00lbk6g";
  };

  buildInputs = [ ghc X11 ]; [3]

  configurePhase = '' [4]
    substituteInPlace XMonad/Core.hs --replace \
      '"xmessage"' '"${xmessage}/bin/xmessage"' [5]
    ghc --make Setup.lhs
    ./Setup configure --prefix="$out" [6]
  '';

  buildPhase = ''
    ./Setup build
  '';

  installPhase = ''
    ./Setup copy
    ./Setup register --gen-script
  '';

  meta = { [7]
    description = "A tiling window manager for X";
  };
})
```

Figure 3.5: Nix expression for xmonad; Source: [DLP10]

through the use of profiles. *Italicized* text represents symlinks, with dotted lines representing their targets. We observe that the user Alice utilizes a symlink to the user-environment at Figure 3.6 point [11]. This user environment subsequently contains symlinks to packages to be used within that environment. Atomic operations within the Nix Store are thus accomplished by switching to various symlinked environments, eliminating inconsistencies in the dependency structure of a package as is often the case with imperative package managers. That is to say, upgrading or rolling back changes is as simple as updating a symlink in the current users $PATH environment variable [DLP10].

Sharing

Software distribution is a key component of package managers, and the ability to

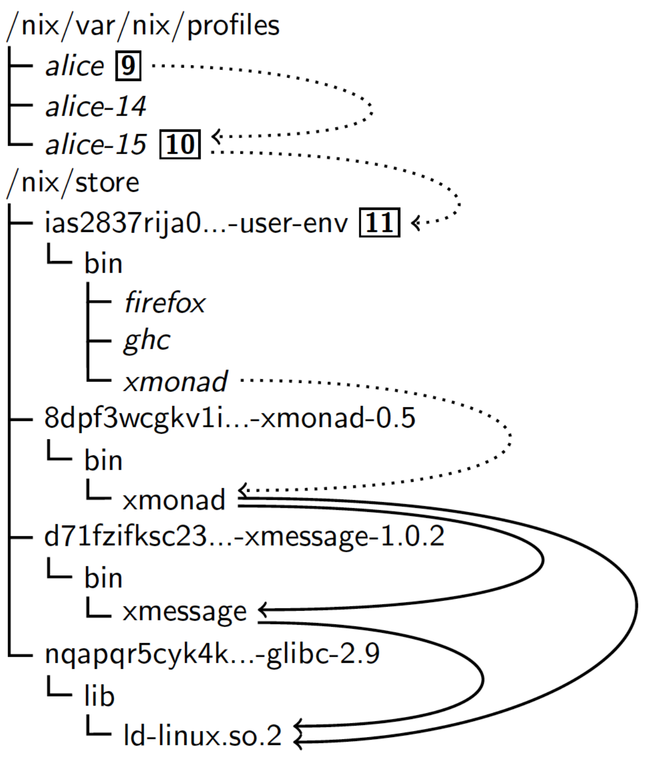


Figure 3.6: The Nix Store: Example for xmonad; Source: [DLP10]

identify software is critical in accomplishing this task. We know from our above descriptions of Nix expressions and the Nix Store that built packages are placed in directories comprised of 160-bit hashes. Because the hash of an individual package is based on that packages derivation, the hash acts as a globally unique identifier that allows for the deterministic identification of a package [DdJV04][DLP10].

The availability of a globally unique identifier provides several advantages over a traditional imperative package manager. Although we have previously described the methods in which the Nix Store builds packages, the functional manner in which they are built also allows for the distribution of binary images. Upon the request of a package, the Nix Store will automatically detect if the (re)building of the package is required (due to for example a configuration change), and when it is not required, will fetch a pre-built binary from a remote repository. Users retrieving packages in this way can be sure that the binaries they are receiving are exactly what they are requesting.

3.1.3 Functional Linux

In section 3.1.2 we saw how the Nix Package manager allows for both the creation and distribution of software packages through the use of functional paradigms. While the Nix Package Manager can be utilized across a wide variety of OSs such as Mac OSX, FreeBSD, and Windows, its core concepts allow for much more powerful applications. In this section, we take the Nix Package Manager further by exploring how its concepts can be used to implement an entire operating system, namely, the NixOS [DLP10].

The Nix derivations used within Nix expressions can also be used to build static parts of an OS such as configuration files that are not modified dynamically at runtime. In Figure 3.7 we can see how one can use a Nix derivation to build an sshd configuration file that will exist in the Nix Store with its own unique hash identifier as described in section 3.1.2. By examining the derivation, we can see that if a user desires that X11 forwarding be enabled, specifically if the option *services.sshd.forwardX11* is set to yes, then the *xauth* package is required. This represents a unique characteristic of the NixOS: it allows for configuration files to enumerate their software package dependencies. A consequence of this is that packages will not be built if they are not referenced by a configuration file, and will not be garbage collected if they are [DLP10].

```
pkgs.writeText "sshd_config" ''
  UsePAM yes
  ${if config.services.sshd.forwardX11 then ''
    X11Forwarding yes
    XAuthLocation ${pkgs.xorg.xauth}/bin/xauth
  '' else ''
    X11Forwarding no
  ''}
''
```

Figure 3.7: Nix Expression to build sshd_config; Source: [DLP10]

To specify the configuration to be used to build the NixOS, the user must edit the */etc/nixos/configuration.nix* file, an example of which can be seen in Figure 3.8. We see that this configuration file specifies what kernel modules should be installed (point [14]), what file systems should be mounted (point [15]), and that the SSH daemon should be enabled (point [18]). We also see the enabling of the *services.sshd.forwardX11* option at point [19], which in turn is utilized by the Nix expression in Figure 3.7 to build the SSHD configuration file [DLP10].

Changes to the NixOS configuration file can be activated through the use of the *nixos-rebuild switch* command. In fact, this command allows the user to perform a number of significant operations, including creating test configurations, as well as rolling back and upgrading the NixOS. These operations share many of the same properties found in the Nix Package Manager described in section 3.1.2: they are performed atomically, they are non-destructive, and they allow system configurations to be reproducible. Figure 3.9 shows how previous configurations are presented to the user, allowing rolling back to be as simple as selecting the desired configuration [DLP10].

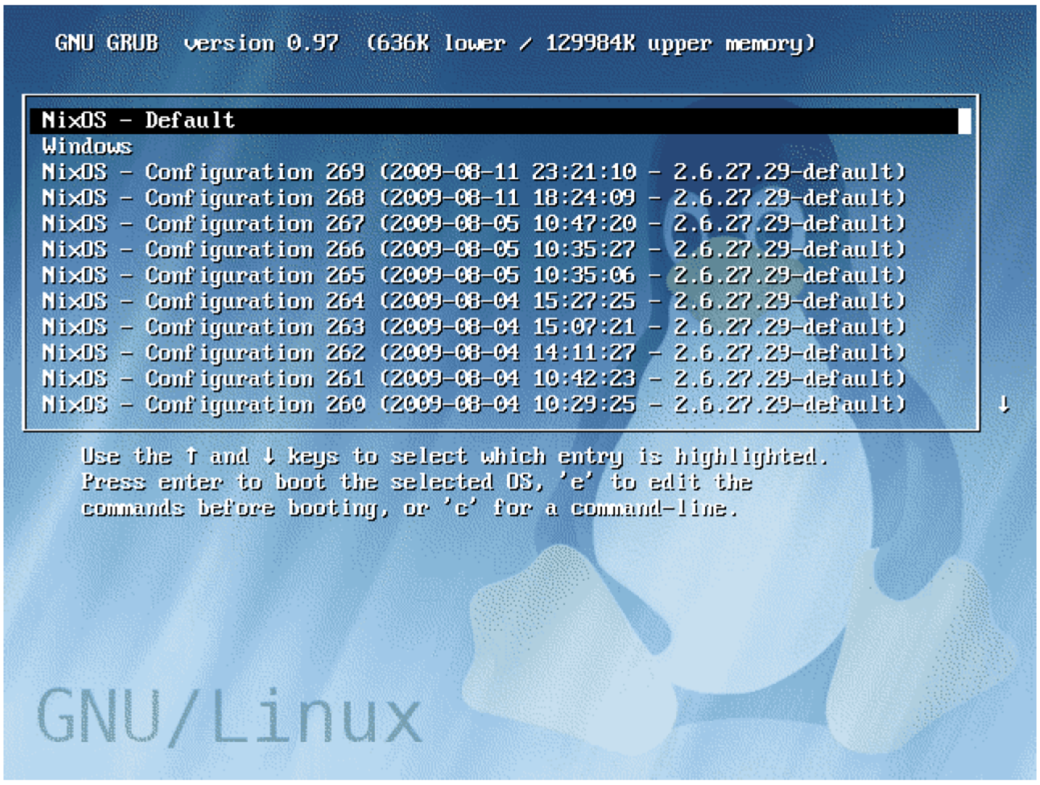
```
{ config, pkgs, ... }: 12

{
  boot.loader.grub.device = "/dev/sda"; 13
  boot.kernelModules = [ "fuse" "kvm-intel" ]; 14

  fileSystems = 15
    [ { mountPoint = "/";
        device = "/dev/disk/by-label/nixos";
      }
      { mountPoint = "/home";
        device = "/dev/disk/by-label/home";
      }
    ];

  swapDevices = [ { device = "/dev/disk/by-label/swap"; } ]; 16

  environment.systemPackages = [ pkgs.firefox ]; 17

  services.sshd.enable = true; 18
  services.sshd.forwardX11 = true; 19

  services.xserver.enable = true; 20
  services.xserver.videoDriver = "nvidia";
  services.xserver.desktopManager.kde4.enable = true;
}
```

Figure 3.8: Nix Configuration File Example; Source: [DLP10]

3.1.4 Version Control Software

In section 2.1.2 we listed four rules that aim to help guide researchers towards creating more useful and reproducible research. A key component to the presented work is captured by Rule 4: Version Control All Custom Scripts. To do so, one must use Version Control Software (VCS) such as Git or Subversion (SVN). In this section, we briefly explain at a high-level the main working components and features of Git, a VCS, that are relevant to the presented work.

VCS allows for the automated management of changes to documents, programs, and collections of information in general. Specifically, it allows the iterative changes to such resources to be tracked over time, and most importantly, for old versions to be retrieved at a later time. In large projects, these features are essential for tasks such as

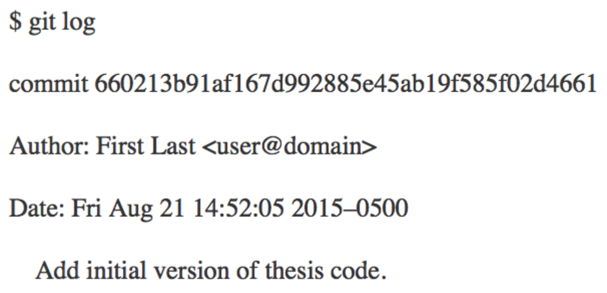


```
GNU GRUB  version 0.97  (636K lower / 129984K upper memory)

NixOS - Default
Windows
NixOS - Configuration 269 (2009-08-11 23:21:10 - 2.6.27.29-default)
NixOS - Configuration 268 (2009-08-11 18:24:09 - 2.6.27.29-default)
NixOS - Configuration 267 (2009-08-05 10:47:20 - 2.6.27.29-default)
NixOS - Configuration 266 (2009-08-05 10:35:27 - 2.6.27.29-default)
NixOS - Configuration 265 (2009-08-05 10:35:06 - 2.6.27.29-default)
NixOS - Configuration 264 (2009-08-04 15:27:25 - 2.6.27.29-default)
NixOS - Configuration 263 (2009-08-04 15:07:21 - 2.6.27.29-default)
NixOS - Configuration 262 (2009-08-04 14:11:27 - 2.6.27.29-default)
NixOS - Configuration 261 (2009-08-04 10:42:23 - 2.6.27.29-default)
NixOS - Configuration 260 (2009-08-04 10:29:25 - 2.6.27.29-default)   ↓

    Use the ↑ and ↓ keys to select which entry is highlighted.
    Press enter to boot the selected OS, 'e' to edit the
    commands before booting, or 'c' for a command-line.
```

Figure 3.9: The GRUB boot menu for a NixOS machine; Source: [DLP10]

finding and fixing bugs, rolling back to working versions of resources, and collaborative work between teams of developers. [CC14].

In order to track changes, Git utilizes a data store called a repository found within the working directory (as *./git*) which a user wishes to version control. After initializing the repository, users are provided with tools to manage the revision history of all objects and files contained within the directory to which the repository belongs. The repository contains information regarding the current version of a given file (or files), as well as references to previous versions of said file(s). To track a file, a user must first add the file to what is called the staging area, and then commit the file. Upon committing, a 40 character Secure Hash Algorithm 1 (SHA-1) hash is generated that allows for the identification and retrieval of the commit, and subsequently its contents, at a later time. This generated hash will prove crucial in the presented work, detailed in section 4.3.5. In Figure 3.10 we see an example of the 'git log' command, containing information about commits within the repository, and specifically the generated SHA-1 hash. Additional parameters may be passed to this command to detail the contained files [BDW16].



```
$ git log

commit 660213b91af167d992885e45ab19f585f02d4661

Author: First Last <user@domain>

Date: Fri Aug 21 14:52:05 2015–0500

   Add initial version of thesis code.
```

Figure 3.10: 'git log' Example; Source: [BDW16]

3.1.5 The Linux Audit Daemon

Crucial in capturing the provenance of data is the monitoring of the system on which the data is utilized and manipulated. Specifically, the system calls, or syscalls for short, must be captured in order to take account of the operations being performed on the data. Syscalls allow for applications in user-space to interface to primitive functions in the linux kernel, and examples include open, close, write, and create. While there are many utilities that allow one to monitor the syscalls invoked on the system, we focus here on The Linux Audit Daemon, or auditd for short [Red].

Auditd is a powerful monitoring/auditing utility for Linux providing users a highly configurable method in which to capture syscalls. There are two main components of auditd that allow for the fine tuning of the monitoring to be performed: the auditd.conf configuration file [GRc], and the audit.rules rules file [GRd]. We briefly cover the functionality that each provides, and in section 4.3.2 we will describe their solution specific implementations [Red].

The auditd.conf file allows users to specify a number of configuration parameters related to the handling of the log files created by the daemon. For the sake of brevity, we describe here only the select few which play important roles in the daemons functionality. [Red].

num_logs
: As one can imagine, the monitoring of an entire system can result in very large quantities of data being collected. This parameter allows the user to configure the maximum amount of log files that can be created. The value specified is very closely tied to the following two parameters.

max_log_file
: This parameters allows for the configuring of the maximum size of an individual log file. This value is specified in terms of megabytes.

max_log_file_action
: Utilizing the previous two parameters, the max_log_file_action parameter de-

scribes what action to perform upon a log file reaching the maximum size as specified by the max_log_file parameter. Its valid values are *ignore, syslog, suspend, rotate* and *keep_logs.* To better understand this parameter, we give an example utilizing the *rotate* option:

```
num_logs = 5
max_log_file = 5
max_log_file_action = rotate
```

In the above example, the host system is limited to five log files, each with a maximum size of five megabytes, being rotated when maximums are reached. The keyword *rotated* specifies that when the maximum number of log files is reached, the oldest log is deleted, the remaining log files are shifted down numerically, and a new log file is created.

There are many more configurable parameters available to the user to define things like the flushing of the log buffers, the name and location of the log files, system disk space actions, and even TCP client parameters for logging to remote peers. The readers are encouraged to read the auditd manpage for more information [Red][GRc].

The second main component of auditd is the audit.rules file. This file allows the user to specify rules and filters dictating what exactly should be monitored by the daemon. At the highest level, users can specify three types of rules: control, file, and syscall. We focus here on syscall rules, as they are the most relevant to the presented work. Syscall rules come in the following form and are arguments to the auditctl command:

```
auditctl -a action,list -F arch=value -S syscall -F field=value -k keyname
```

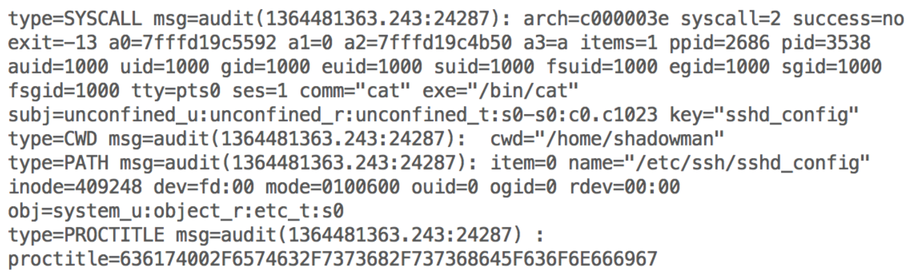
The **-a** argument specifies that the following rule be appended to the current list. The **action** keyword may take only the value *always* or *never*, and indicates when to enforce the rule for the specified **list**. The Linux kernel provides five rule matching lists, though here we explain only the *exit* list and leave the remaining for the reader. The *exit* list is checked at the time a syscall exits, and provides the daemon with the largest amount of information, or fields, about the exiting syscall [Red][GRd].

The **-F** argument allows a user to specify filtering options for the captured data. It is required to indicate the host system's architecture before the syscall that is to be monitored. Possible values for **arch** are *b32* and *b64* for 32-bit and 64-bit host systems respectively. After having specified the host's system architecture, the syscall to monitor is provided by the **-S** argument. Further filtering options can be then provided and allow data to be filtered by fields related to a user's ID, the success of a syscall, file types, and message types [Red][GRd].

Finally, the **-k** options allows users to label log messages matching the specified rule with a key. The key can be an arbitrary string up to 31 bytes long, and allows for the easier identification of log messages that pertain to a specific activity a user wants to monitor. There are many more configurable options for auditd to further refine the rules, and we encourage the readers to review the auditctl man-page [GRa].

System events are captured by auditd in the form of records. In Figure 3.11, we see an example of a captured event consisting of four record types: SYSCALL, PATH, CWD, and PROCTITLE. Records of the SYSCALL type provide information about the called syscall such as the caller's User ID (UID), the syscall's exit value, the command name that called it, and the path to the calling executable. PATH records provide the path to the argument(s) passed to the syscall. CWD records provide the current working directory (CWD) for the process that invoked the syscall, and PROCTITLE records encode the entire executed command-line in hex that lead to the captured event.

The records seen in 3.11 detail the execution of the */bin/cat* command. Specifically, the records indicate that the */bin/cat* command was executed from the */home/shadowman* directory, and was used to print the contents of the file */etc/ssh/sshd_config*. The time of execution (1364481363.243:24287) is represented in the Unix time format, and can be converted to a human-readable date with the Linux *date* command. Additionally, the arguments provided to the syscall are found in the fields *a0*, *a1*, *a2*, and *a3* in hexidecimal. These arguments consist of the file to be accessed, mandatory flags governing the mode of access (read-only, write-only, read/write), and additional optional flags further defining the syscalls exact operation. We note that these arguments are not to be confused with those passed to the */bin/cat* command. Finally, we see the hexidecimal representation (636174002F6574632F7373682F737368645F636F6E666967) of the full command line entry used to execute this command.



```
type=SYSCALL msg=audit(1364481363.243:24287): arch=c000003e syscall=2 success=no
exit=-13 a0=7fffd19c5592 a1=0 a2=7fffd19c4b50 a3=a items=1 ppid=2686 pid=3538
auid=1000 uid=1000 gid=1000 euid=1000 suid=1000 fsuid=1000 egid=1000 sgid=1000
fsgid=1000 tty=pts0 ses=1 comm="cat" exe="/bin/cat"
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key="sshd_config"
type=CWD msg=audit(1364481363.243:24287):  cwd="/home/shadowman"
type=PATH msg=audit(1364481363.243:24287): item=0 name="/etc/ssh/sshd_config"
inode=409248 dev=fd:00 mode=0100600 ouid=0 ogid=0 rdev=00:00
obj=system_u:object_r:etc_t:s0
type=PROCTITLE msg=audit(1364481363.243:24287) :
proctitle=636174002F6574632F7373682F737368645F636F6E666967
```

Figure 3.11: Auditd Log Example; Source: [Red]

3.2 Data Interaction and Visualization

This section briefly outlines two tools that allow users to interact with provenance information represented by RDF graphs or OWL semantics. These tools will be used to explore the data of the working examples defined in section 3.3 and studied in 4.4.

3.2.1 OntoGraf

OntoGraf [Fal16] is a plugin for Stanford's Protégé application [Mus15]. OntoGraf makes it possible to interactively navigate ontologyies by virtue of a graphical representation of the ontology's individuals and their relationships. Users are able to search ontologies for individuals by keyword, and filter results by relationship and individual type.

In Figure 3.12 we see an example query for the keyword "cheeseypizza". The resulting interactive graph shows the ontology's classes related to the searched keyword, as well as the relationship(s) between them. That is, users can expand and collapse nodes to explore the relationships represented in the ontology. Additionally, one can search for terms in a variety of ways: contains, starts with, ends with, exact match, and regex expressions. Finally, OntoGraf provides multiple visualization styles that organize the results for better readability as desired.

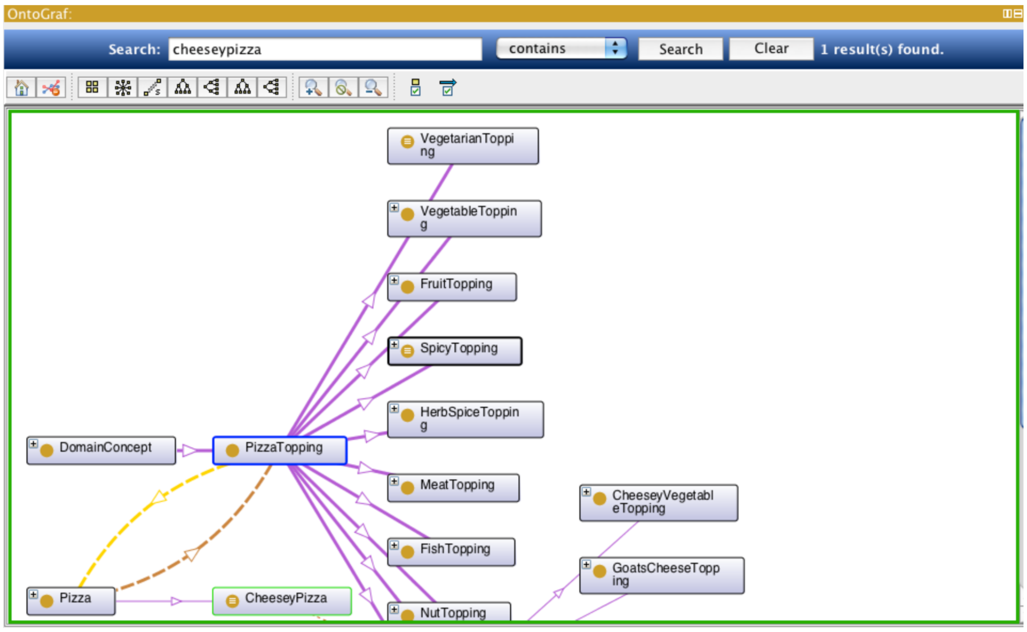
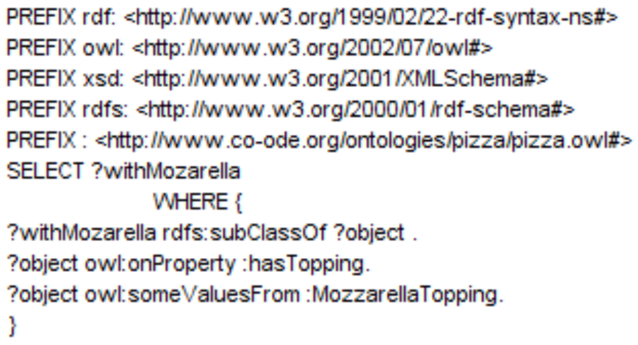


Figure 3.12: OntoGraf Example; Source: [Fal16]

3.2.2 SPARQL Query

The SPARQL Query Language allows users to query RDF graphs as well as OWL Ontologies. Compared to OntoGraf, SPARQL Queries provide finer search granularity. Ontology prefixes, property values, and relationships can be queried for, with the results being represented by sets or RDF graphs [GSP13]. For convenience, we again use a plugin for Stanford's Protégé application, SPARQL Query [Red16]. SPARQL queries follow a very traditional query structure, with modifications to allow the specification of RDF triples and semantic web prefixes [GSP13].

In Figure 3.13 we see an example of a SPARQL Query. We note that the prefixes are specified at the beginning of the query. Namespaces utilized in the ontology to be queried must be specified at the beginning of the query by the PREFIX keyword. A SPARQL Query can be composed in four different forms: SELECT, CONSTRUCT, ASK, and DESCRIBE. In Figure 3.13 the SELECT form is being used, and we can see the previously specified prefixes being used in the building of triples. The shown example will find all pizzas that have mozzarella cheese as a topping [GSP13].



```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX : <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
SELECT ?withMozarella
            WHERE {
?withMozarella rdfs:subClassOf ?object .
?object owl:onProperty :hasTopping.
?object owl:someValuesFrom :MozzarellaTopping.
}
```

Figure 3.13: SPARQL Query Example; Source: [Red16]

3.3 Working Examples

In this section we define two working examples on which we will utilize the solution as outlined in Chapter 4.

For our first working example, we will present the ontology resulting from the captured provenance for the invocation of a single process. Specifically, we will use the "pdflatex" terminal command to generate a PDF file from a TEX file. It is important to note here that the solution is not restricted to be only functional for terminal commands. By nature of the Linux Audit Daemon, all syscalls, even those initiated via Graphical User Interfaces (GUIs), can be captured and processed.

The chosen TEX file, DatabaseReport.tex, requires the inclusion of numerous images and libraries, and when compiled with pdflatex, allows for the general functionality of the solution system to be demonstrated. The terminal command "pdflatex DatabaseReport.tex" will be executed by our test user account, testResearcher1. At the time of execution, our solution will be running in the background, spawned by the root user of the host machine.

Once the pdflatex process has completed and the provenance information added to the ontology, the data can be visualized and interacted with (as seen in section 4.4) by way of the Protégé plugins OntoGraf and SQARQL query as detailed previously in sections 3.2.1 and 3.2.2.

After having established an understanding of the general functionality of the presented solution system, we provide a less trivial working example. In this working example we capture the provenance of a Latex document beginning from the time of its acquisition, and continuing through numerous edits and compilations. During this process, edits to the document will include both textual and included resources, and updates to various components of the NixOS will be performed. Doing so will allow us to present in further detail how the functionality provided by the Git Version Control and NixOS components is represented in the resulting ontology. To bring the work presented in this thesis full circle, the document used in this working example will be the very LaTeX document used to compile this thesis itself.

3.4 Summary

In this chapter we covered the main technologies, concepts, and components that make up the presented prototype tool. Section 3.1 introduced us to The Semantic Web, where formalized structure brings form and relationships to the data found on the WWW. In section 3.1.1 we saw how the various layers of The Semantic Web, specifically the RDF and Ontology layers, allowed for great precision in the specification of data types of relationships. We furthered defined how the OWL2 Ontology Language and PROV-O can be used together to represent data provenance.

Section 3.1.2 provided us with our first look at one of the core components of the presented prototype tool: the Nix Package Manager. We defined the shortcomings of traditional imperative package managers, and how the Nix Package Manager solves many of these problems by employing a functional model. We also saw how Nix Expressions and the Nix store play key roles in the building and sharing of software packages.

In section 3.1.4 we saw how VCSs can be used to to track file changes and to identify specific revisions of files. This functionality was examined in GIT, and will play a crucial role in the identification of data and its provenance data in the presented prototype tool.

Functional Linux, in particular the NixOS, was detailed in section 3.1.3 and we saw how the fundamental concepts of the Nix Package Manager can be used to create a functional operating system. The NixOS will provide an identifiable and reproducible working environment in which workflows and experiments can be executed.

Section 3.1.5 gave insight into the Linux Audit Daemon and how it will provide us with the provenance information of data used by the system. We saw how it can be configured and used to trace the syscalls executed on the host system.

Provenance visualization utilities such as SPARQL Queries and OntoGraf were explored in section 3.2, and finally, in section 3.3 we introduced the use-case that will be studied in Chapter 4.

CHAPTER 4

Ontological Representation of Provenance

In the previous chapter, we covered the methodology used in the presented prototype tool. This chapter explores the solution in detail, including how the previously explored concepts and components are tailored to suit the solution. Furthermore, we will see how the working examples in section 3.3 expose the functionality of the presented solution. Lastly, we will examine the performance of the solution system and its complexities. To remain true to our effort to promote the reproducibility of computation research, the code used to implement the work in this thesis can be retrieved from the following GitHub repository: *https://github.com/RothTuThesis/ProvenanceOntology*

4.1 Solution Outline

The goal of the prototype tool is to allow users and researchers alike to better the reproducibility of their work and findings. We accomplish this by providing a host system that automatically captures provenance information without explicit interaction from the user or extensive programming or Information Technology (IT) knowledge. Additionally, captured provenance information can then be queried for in a systematic manner, allowing the user quick and easy access to the provenance for any given file contained within the monitored directories. In Figure 4.1 we see an outline for the prototype tool presented in this work.

Through the combined functionality of the components seen in Figure 4.1, users must only start the auditd process as desired, or simply leave it running in the background. The Log Parser automatically monitors the log file and will (though the combined use of the Git Repository and Nix Store) enter the information into a PROV-O ontology. The ontology can then be queried at a later time via Stanford's Protégé and its various plugins such as SPARQL Query and OntoGraf.

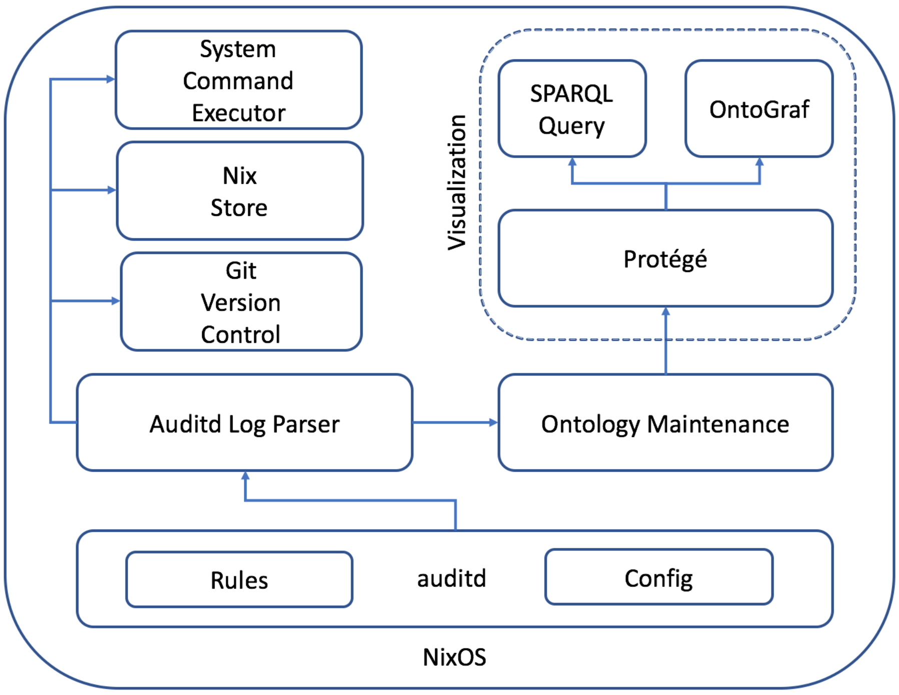


Figure 4.1: Solution Structure

We see that the prototype tool is not only a tool that one can install and use on an existing system, but is rather an entire system solution with the NixOS at its core. We note here that the aim of the solution system is not to create a VM which can/must be delivered to other researchers to enable research reproducibility. Instead the solution is intended to provide researchers with an OS on which researchers can perform experiments without the additional overhead or concern of manually tracing data's provenances or creating explicit workflows.

The following sections expand on the components seen in Figure 4.1 and outlined in Chapter 3.1. By the end of this chapter, the reader will be familiar with how these components are assembled and configured and how they provide the structure and functionality of the solution.

4.2 Resources

4.2.1 OWL2

The OWL2 Ontology Language is one of the pillars of the presented solution, as it allows for data to be interpreted and processed, as opposed to nearly presented to the user. Within the scope of this work, the OWL2 API is used to both manipulate and interpret the classes (and their members) and relationships defined by PROV-O (and subsequently the OWL2 Ontology Language). In this section we focus on the main (relevant) features of the OWL2 Ontology Language and how the OWL2 API provides access to these features. Specifically, we look at entities, literals, classes, data properties, object properties, and axioms. [PSPM12]

The fundamental building blocks of an OWL2 Ontology are entities. Entities allow us to define the vocabulary of an ontology and are identified by a unique IRI. Additionally, the data properties, object properties, and classes, are all entity subtypes. Classes allow for the representation of sets of individuals, while object and data properties allow one to specify relationships with other individuals/classes and data values respectively. We will see in section 4.3.7 how PROV-O defines its vocabulary as relevant to the presented work. [PSPM12]

The data values specified by data properties are captured by literals. Literals allow for the specification of string values, along with a datatype that indicates how to interpret the string. For example, a literal can be of type *xsd:string*, indicating that it is a character string in Extensible Markup Language (XML) [BM04].

Axioms allow one to specify what is true in the domain in which the ontology exists. For the purposes of the presented work, axioms can be defined for classes, as well as both object and data properties. Example of class axioms include SubClassOf, EquivalentClasses, and DisjointUnion. Examples of object property axioms include DisjointObjectProperties, InverseObjectProperties, and SymmetricObjectProperty. And finally, examples of data property axioms include DataPropertyRange, EquivalentDataProperties, and DisjointDataProperties. We note that PROV-O employs its own axioms that build upon those of OWL2, and will be covered in section 4.3.7. [MP12] [LMS13].

In the presented solution, we access the above functionality by way of a few key functions provided by the OWL2 API:

- OWLDataPropertyAssertionAxiom getOWLDataPropertyAssertionAxiom()
- OWLObjectPropertyAssertionAxiom getOWLObjectPropertyAssertionAxiom()
- OWLLiteral getOWLLiteral()
- void addAxiom()

However, to use the first three functions, we must instantiate an OWLDataFactory, and for the last function, we must instantiate an OWLOntologyManager. The OWLOntologyManager provides the main access point for actions such as creating, loading, and accessing an ontology. The OWLOntologyManager is in turn required to instantiate the OWLDataFactory, which provides the functionality required for creating entities, axioms, and classes. Further details on how these functions are utilized in the presented solution can be found in section 4.3.4. [HB09]

4.2.2 JGit Java Library

To access (most of the) Git functionality in our solution, we utilize the JGit Java Library [The17]. In the presented solution, JGit is utilized to perform basic Git operations such as creating a Git Repository, as well as adding and committing files to the repository. However, in the course of designing the presented solution, certain Git functionalities were better served by the standard Git terminal commands. For example, the status of the Git repository as returned by the JGit library proved inconsistent and unreliable. As a result, specific Git terminal commands were used and are detailed in sections 4.2.3.

4.2.3 Unix Commands

Not all the required functionality could be implemented in Java. As a result, certain system commands needed to be executed on the host system. We briefly outline the used system and Git commands, and in section 4.3.6 we detail how they are incorporated into the presented solution.

readlink

The readlink command allows the user to resolve the true value/directory of a symbolic link. The NixOS relies heavily on symbolic links and as such, the log files from auditd contains many of them as well. Resolving the symbolic links allows the presented solution to obtain the full directory of the used resources and files. When the used resource is something stored in the Nix Store, the resolved directory contains the 160 character hash, thus allowing precise identification and more accurate provenance information.

awk

The awk command is a general purpose pattern scanning and processing utility for textual data modification. In the scope of the presented work, the awk command is used simply to extract the relevant fields from textual data. Specifically, it is used to retrieve the username for a particular UID, as the auditd log files contain only UIDs and not usernames. The returned username is later used in the ontology to relate users to processes executed on the host system.

git [log | rev-parse | diff-index | update-index]

These various git commands are used to provide functionality that could not be reliably provided utilized via JGit. The *diff-index* and *update-index* commands are used in conjunction to obtain the status of the Git repository, while the *log* and *rev-parse* commands are used in conjunction to retrieve a unique SHA-1 for each tracked object/file. [Tor17]

4.3 Component Implementation

4.3.1 NixOS

The backbone of the solution system is the NixOS. In section 3.1.3 we explored the main concepts that make the NixOS what it is, and in this section, we detail how it is used in the solution system.

Although the exact NixOS version is not required for future replication of the solution, the utilized version is 17.09.1756.c99239bca0. More important is the configuration file seen in Figure 4.2. We see a handful of important sections in this configuration file. First, we note that the *imports* section provides several Nix derivations critical to the functionality of the NixOS. In particular, these imports provide the resources allowing for the NixOS to be ran in a graphical environment, to be packaged and used as a virtual box image, to allow its configuration(s) to be modified and rebuilt, and finally, to subscribe to the necessary Nix Store channel(s) for package manager functionality.

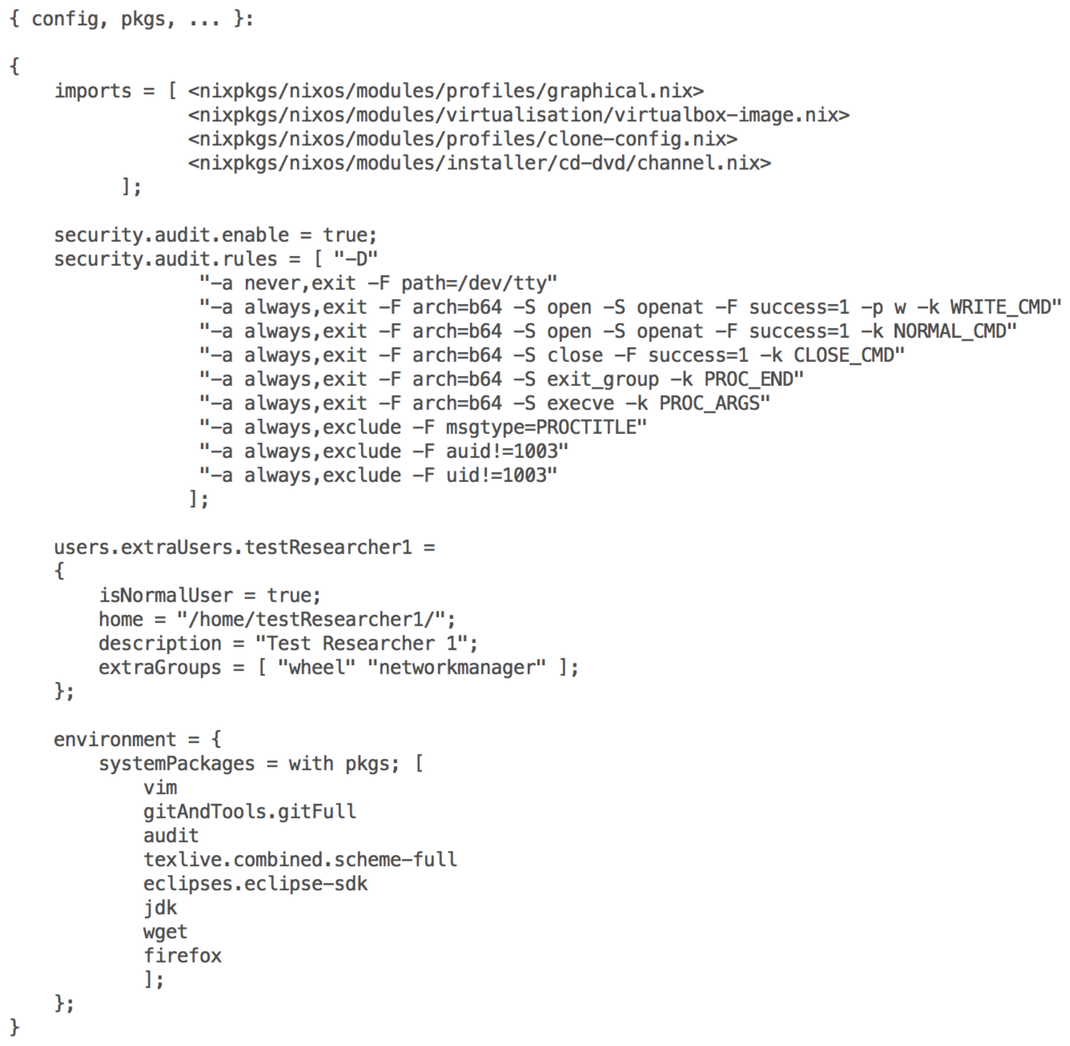
The next section specifies the audit rules, which we will visit in section 4.3.2. After the audit rules are specified, we see the description of our test user account, testResearcher1. The user's home directory, description, and privileges are specified. Next, we see the inclusion of the system packages to be installed on the NixOS. While not all of the specified packages are critical to the implementation of the solution system, of particular importance are the "git" and "audit" packages. We note that that the "audit" package contains auditd, as well as many additional utilities allowing for the modification of and interaction with the system's audit entries. The remaining packages to be installed are specified to provide the functionality required to demonstrate the applicability of the presented solution system via the working examples described in section 4.4.

4.3.2 The Linux Audit Daemon

The Linux Audit Daemon, or auditd, is responsible for logging all the required provenance information of objects on the host system. In this section, we cover the implemented auditd rules and the configuration that governs its operation.

The auditd rules can be seen in Figure 4.2. We briefly explain the additions to the syntax not covered in section 3.1.5. First, the *audit* service is set to enabled, after which the rules to be used are specified. To ensure the correct rules (including the ordering) are specified, the existing rules are deleted with the "-D" parameter. The following four rules are all very similar in that that specify syscalls to monitor. Each rule is indicated to be for 64-bit systems, and to only log calls that are successful. We see that the first rule monitors the *open* and *openat* syscalls, and contains the additional "-p" parameter. This parameter allows us to specify the access type to monitor. Here we specifically monitor the objects which are written to by passing "w" to the "-p" parameter.

To aid the process of parsing the log file entries, the rule entries are given a unique label for identification. In the solution's auditd rules, these keys are WRITE_CMD,



```
{ config, pkgs, ... }:

{
    imports = [ <nixpkgs/nixos/modules/profiles/graphical.nix>
                <nixpkgs/nixos/modules/virtualisation/virtualbox-image.nix>
                <nixpkgs/nixos/modules/profiles/clone-config.nix>
                <nixpkgs/nixos/modules/installer/cd-dvd/channel.nix>
          ];

    security.audit.enable = true;
    security.audit.rules = [ "-D"
                 "-a never,exit -F path=/dev/tty"
                 "-a always,exit -F arch=b64 -S open -S openat -F success=1 -p w -k WRITE_CMD"
                 "-a always,exit -F arch=b64 -S open -S openat -F success=1 -k NORMAL_CMD"
                 "-a always,exit -F arch=b64 -S close -F success=1 -k CLOSE_CMD"
                 "-a always,exit -F arch=b64 -S exit_group -k PROC_END"
                 "-a always,exit -F arch=b64 -S execve -k PROC_ARGS"
                 "-a always,exclude -F msgtype=PROCTITLE"
                 "-a always,exclude -F auid!=1003"
                 "-a always,exclude -F uid!=1003"
                ];

    users.extraUsers.testResearcher1 =
    {
        isNormalUser = true;
        home = "/home/testResearcher1/";
        description = "Test Researcher 1";
        extraGroups = [ "wheel" "networkmanager" ];
    };

    environment = {
        systemPackages = with pkgs; [
            vim
            gitAndTools.gitFull
            audit
            texlive.combined.scheme-full
            eclipses.eclipse-sdk
            jdk
            wget
            firefox
            ];
    };
}
```

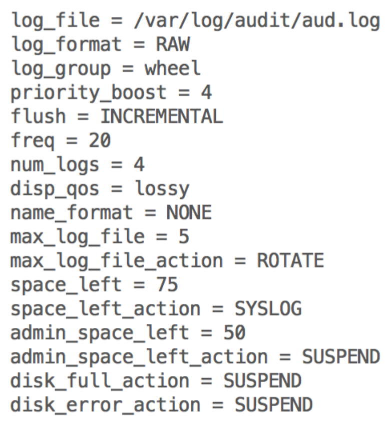
Figure 4.2: NixOS Solution Configuration

NORMAL_CMD, CLOSE_CMD, PROC_ARGS, and PROC_END, and are used to key entries matching syscalls that write to files, open files, close files, provide the arguments with which a process was invoked, and end processes respectively. The order ordering of the rules is critical to the proper configuration of auditd. Syscalls that match multiple auditd rules will be keyed to only the first matching rule the syscall is checked against. In Figure 4.2, if the NORMAL_CMD rule to monitor the *open* and *openat* syscalls (without the write permission restriction) would be placed first in the list, all *open* syscalls would match only that rule entry, and none the entry monitoring the write access types. This is especially critical in the implementation of the algorithm(s) used to process the log entries for provenance information.

The last three rules specify the exclusion of log entries. We exclude here all log entries with "msgtype=PROCTITLE", as the information it provides is not necessary and often redundant. While traditionally this msgtype captures the full command-line used to invoke a process, log entries of this type corresponding to processes invoked from shells such as "bash" fail to capture this information. In such cases, the exclusion of this entry type serves to maintain a smaller log file. Additionally, when this entry type does capture the command-line information, it is redundant in that the entries for the *execve* syscall, subsequently matching the label PROV_ARGS, also contain this information. Therefore, we chose to capture only the more comprehensive *execve* syscall.

Additionally, we note that the UID of our testResearcher1 user is 1003. The final two exclusion rules exclude all entries not originating from the Audit User ID (AUID) (the logged in user) and UID (the user who started the analyzed process) from this UID. The UID should be modified with respect to the user accounts for which the researcher desires to capture provenance information [GRa]. These three exclusion rules are critical in the feasibility of the solution in that they prevent the generation of an unmanageable amount of log entries. These additional log entries originate from the solution itself performing the parsing of the entries. Thus, to solve this problem, the application must be ran from a different account than that on which the processes to monitor are run.

In Figure 4.3 we see the solution's audit configuration file. In it, we specify a number of configurable options such as the location of the resulting log file, the actions to be taken in the event of low system space, the number of logs, and the rotating of the logs. Importantly we see that in our solution system, a log file will be deleted after 20 MB have been written to the system, and then every time an additional 5 MB have been written to the system. We leave it to the reader to explore the configuration in further detail. [GRc].



```
log_file = /var/log/audit/aud.log
log_format = RAW
log_group = wheel
priority_boost = 4
flush = INCREMENTAL
freq = 20
num_logs = 4
disp_qos = lossy
name_format = NONE
max_log_file = 5
max_log_file_action = ROTATE
space_left = 75
space_left_action = SYSLOG
admin_space_left = 50
admin_space_left_action = SUSPEND
disk_full_action = SUSPEND
disk_error_action = SUSPEND
```

Figure 4.3: Auditd Solution Configuration (/etc/audit/audit.conf)

4.3.3 Auditd Log Parser

This component of the solution system is responsible for parsing all of the entries auditd makes to the specified log file. It contains all of the algorithms and logic for the collection and processing of the log entries written to the auditd log file. In this section, we detail the working structure of this parsing component and how it interacts with the other components.

In Figure 4.4 we find an overview of the log parser's structure. The core functionality of the parser revolves around its ability to monitor the log file for changes. To do this, we use the Tailer class from the Apache Commons Application Programming Interface (API). The Tailer class is an implementation of the unix "tail -f" command, and allows a Java application to spawn a listener thread signaling when a new entry is written to a file. The new entry is subsequently provided to a *handle()* function, where it is parsed and the necessary actions are performed.

The labeled syscalls entries seen in 4.2, each contain (different) critical information that must be parsed to accurately represent the provenance of a given object and must be handled differently. The PROC_END entry allows us to detect the completion of a process on the host system, while the combination of the CLOSE_CMD and WRITE_CMD entries allow us to detect that files have been modified, and when the modification of said file is complete. Entries matching the label PROC_ARGS provide the command-line used to invoke a process, thus providing the utilized arguments and parameters. And finally, the NORMAL_CMD is a catch all rule for *open* and *openat* syscalls that do not match the first WRITE_CMD.

Depending on the matched rule and the type of log entry, specific actions must be taken, the most important of which is the entering required information into the ontology via calls to the OWL2 API. Details about the ontological representation of this information and its creation can be found in sections 4.3.7 and 4.3.4 respectively. Additionally, the necessary maintenance actions must be performed when a file has been modified, such as calling Git commands to keep the repository up-to-date. Further details on this process can be found in section 4.3.5.

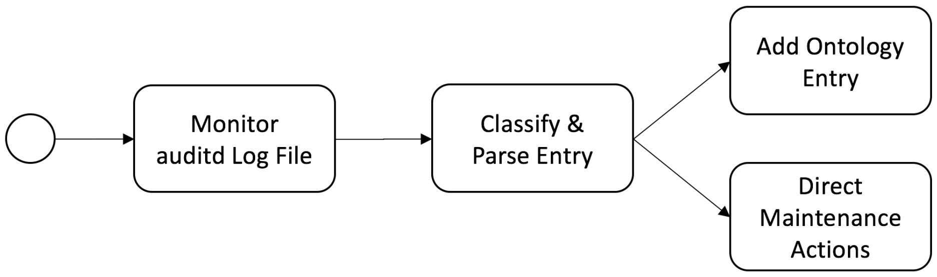


Figure 4.4: Audit Log Parser Solution Structure

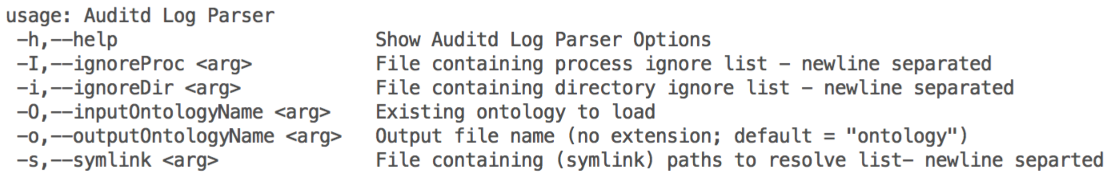
The presented solution system also allows for the customization of how the Auditd Log Parser parses the log entries. In particular, users are able to provide lists of executables and directories to be ignored, as well as symlinks that should be resolved. This information is provided to the Auditd Log Parger via command line arguments as seen in Figure 4.5. The motivation for these features is to allow the user to reduce the size of the ontology by ignoring entries from executables and/or directories whose provenance information is deemed (by the user) not necessary. This often includes background processes/daemons performing system indexing operations, and non data modifying terminal commands such as *ls*,*ps*, and *grep*. It is also often the case that executables query directories such as *proc*, *sys*, and *dev* for process and system information which may not be required for the desired provenance to be captured.

The ability to specify the symlinks to be resolved also allows for not only the size of the resulting ontology to be managed, but also for the performance of the Auditd Log Parser to be streamlined. This stems from the observation that many resources utilized by applications are contained within Nix Store directories. The user is thus able to specify the granularity level of the resulting ontology by configuring the directories whose symlinks should be fully resolved. Additionally, should a directory resolve to a Nix Store location, only the top level directory of the resolved symlink will be stored in the ontology. As resolving symlinks is a timely and expensive operation, the results of previous invocations of this resolution are stored for quick access during subsequent log parsing. Should a user choose not to resolve a given directory in which a large amount of utilized resources reside, while the resulting ontology will provide a more granular depiction of the utilized resources, it will also increase (perhaps undesirably) in size, restricting the ability to graphically interact with it. It is thus necessary for the user to determine the balance of performance and granularity that fits their needs. Examples of this functionality can be seen in section 4.4.

Finally, the handling of the resulting ontology is also configurable. Specifically, users are able to indicate that an entirely new ontology be generated, or that an existing ontology should be loaded and subsequently added to. Additionally, the name of the resulting ontology may be provided, allowing users to chose to either (1) overwrite an existing ontology, or (2) to create a new file for the resulting ontology. This configurability provides users with the ability to capture provenance information for either single process executions, or for extended periods of time while using the solution system. Users also have the ability to export the ontology in its current state without ending Auditd Log Parser process, or exporting the ontology while also ending the process by passing "export" or "exit" to the process.

4.3.4 Ontology Maintenance

This component is responsible for creating the connection between the OWL2 API and PROV-O. That is, it utilizes the OWL2 API to instantiate and maintain the PROV-O ontology. Specifically, this component must create all the classes, data properties, object properties, annotations, and relationships necessary to represent the desired provenance



```
usage: Auditd Log Parser
 -h,--help                      Show Auditd Log Parser Options
 -I,--ignoreProc <arg>          File containing process ignore list - newline separated
 -i,--ignoreDir <arg>           File containing directory ignore list - newline separated
 -O,--inputOntologyName <arg>   Existing ontology to load
 -o,--outputOntologyName <arg>  Output file name (no extension; default = "ontology")
 -s,--symlink <arg>             File containing (symlink) paths to resolve list- newline separted
```

Figure 4.5: Auditd Log Parser Command Line Arguments

information. To properly do so, the Ontology Maintenance component must import all additional ontologies that are utilized by PROV-O and specify the prefix corresponding to each.

Additionally, while it is the Auditd Log Parser that decides what, when, and how an ontology object should be updated, it is the Ontology Maintenance component that directly interacts with the OWL2 API. This is accomplished through five API calls:

- void addOntologyClassAssertionAxiom()
- void addOntologyDataPropertyAssertionAxiom()
- void addOntologyDataPropertyAssertionAxiomLiteral()
- void addOntologyObjectPropertyAssertionAxiom()
- void addAtTimeObjectPropertyAnnotation()

In the presented solution, only two literal datatype will be used: *xsd:dateTime* and *xsd:string* from the XML Schema [BM04]. The *xsd:dateTime* datatype adheres to the formatting guidelines as defined by ISO 8601 [ISO04] and seen below in the following example:

yyyy '-' mm '-' dd 'T' hh ':' mm ':' ss zzzzzz
2017-11-02T16:02:17+01:00

Here, the 'T' is a separator between the indicated date and time, while the 'zzzzzz' specifies the timezone, which in this example is Central European Time (GMT+1). [PSPM12]

The Ontology Maintenance component is also responsible for the loading and saving of ontologies. While an ontology of any format can be loaded, the chosen output format upon saving has been chosen to be the RDF/XML format. This format was chosen as it provides a clear, concise, and minimal representation of the resulting ontology. The resulting file is thus an RDF/XML serialization of the RDF translation of the ontology.

4.3.5 Git Version Control

This component is responsible for initializing and maintaining the Git repository. It does so by interacting with the JGit API as well as the GIT system commands. This component helps ensure a few key concepts crucial to the solution.

First, it is important to note here that the JGit API is capable of handling a traditional ".gitignore" file. As a critical part of the solution system, the ".gitignore" file contains an entry to ignore all hidden files and directories. That is, files and directories that begin with a dot. Second, this component ensures the integrity of the Git repository. Specifically, files are only committed when modified. This may seem trivial, but the JGit API proved not reliable in regard to the repositories status information, thus Git terminal commands must be used to correctly determine the status. Finally, it provides the means in which the Auditd Log Parser can retrieve the unique SHA-1 values for an object contained within the repository. These Git terminal commands are provided by this component in the form of pre-formatted strings to be called by the Auditd Log Parser and executed by the System Command Executor component in section 4.3.6.

Additionally, in the solution system the Git repository is initialized to the user's home directory. In our working examples, this directory is */home/testResearcher1*. Doing so will allow for the provenance information of all files within the user's home directory to be captured and processed, while at the same time excluding this information for anything outside of this directory. Limiting the scope of the repository is important as most of the data outside of the user's home directory is versioned by virtue of the NixOS, and its exclusion gives way to a more feasible solution and provides a more well-defined "workspace".

4.3.6 System Command Executor

The System Command Executor component provides the API necessary to execute terminal commands from a Java application. This component is utilized by the Auditd Log Parser component to execute Git commands and standard unix commands to accomplish tasks that are not directly solvable in Java. For example, to obtain the username that belongs to a UID found in the auditd log file(s), we must utilize the awk command to parse the correct information from the */etc/passwd* file.

Commands are executed within the Bash shell via the CommandExecutor class of the Apache Commons API [Apa17]. The CommandExecutor class allows the Java application to obtain both the output of the executed command, as well as its exit value.

However, executing terminal commands from a Java application is not without its drawbacks, and can often lead to longer than desired processing times. Great effort has been made to utilize only the necessary terminal commands, and to do so in such a way as to have the least impact on the application's runtime.

4.3.7 Prov-O Model

The information parsed by the Auditd Log Parser component and provided to the Ontology Maintenance Component is represented by the model found in Figure 4.6. In it, we see the relationships used to link the various entities, activities, and agents together.

In order to create unique ontology objects, the names of entities and activities are modified with the addition of a unique identifier. For entities and activities beginning with PROCESS_, this identifier is the Message Digest 5 (MD5) hash of the combination of the process's name, its Process ID (PID), and the system time. For the lowest tier entities, this unique identifier is the (unique short) SHA-1 hash value of the object in the solutions Git repository. This unique identifier is not necessary for libraries used by a process, as they reside inside the Nix Store, and thus already include a unique 160-bit hash.

We note here that the *_Collection entities do not represent actual data present on the host system, but rather superficial structures only present in the ontological representation. This design choice in the model allows for the compartmentalization of provenance information, leading to simplified visualizations and better organization. In our case, resources are either used libraries, created files, or used resources other than libraries. Additionally, the *_Collection entities are of type "*prov:collection*", a subclass of the Entity class.

Entities in the ontology representing used resources and created files contain an extra object property annotation and up to two data property assertions to provide more information about the entity. Specifically, the time of use is represented by the *prov:atTime* object property annotation, the time of generation is represented by the *prov:generatedAtTime* data property assertion, and the location (directory) of the entity on the host system is represented by the *prov:location* data property assertion. The location is not part of the entities name in efforts to increase the ontologies readability and utilization by visualization tools. We note that these additions are not appropriate for the library entities, as their names contain both the directory and unique identifier (from the Nix Store) as a design decision.

Libraries are thus represented in the ontology only by their directory. This design choice is in effort to improve readability and keep the size of the ontology minimal, and is only possible due to the nature of the NixOS. That is, because library files reside inside the Nix Store, each can be traced to a unique directory. The Nix Store makes it possible to rebuild (or retrieve) the exact directory the used library is located within via the correct derivation.

4.4 Working Examples

In this section we present our working examples as described in section 3.3. We begin first with section 4.4.1 where we use the pdflatex application to generate a PDF file. This working example will provide an understanding of the basic functionality of

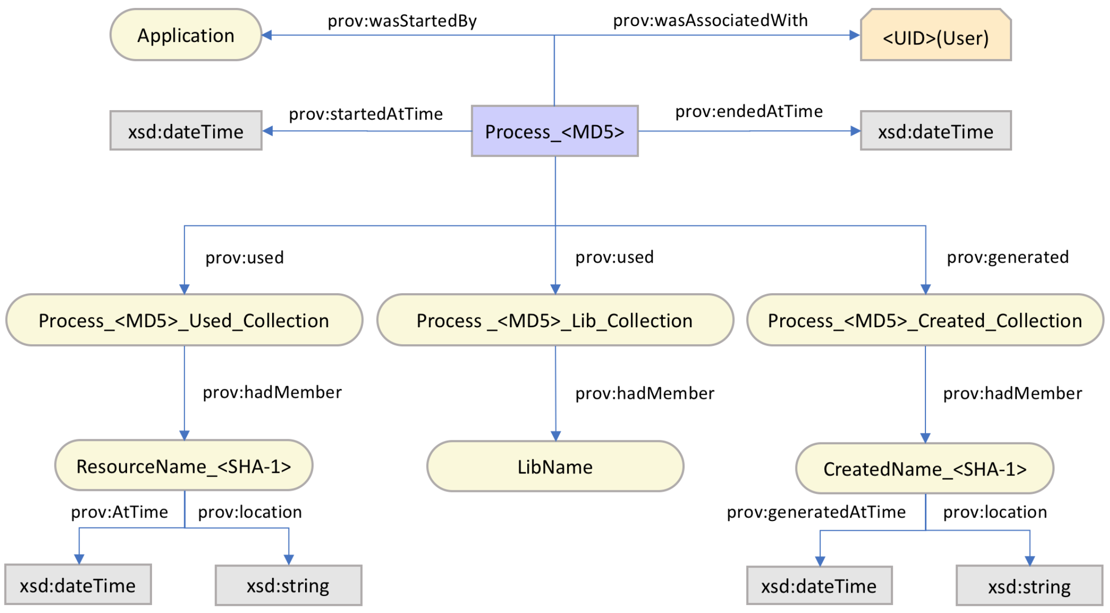


Figure 4.6: Prov-O Solution Structure

the presented solution. Then, in section 4.4.2, we explore the presented solution in more detail by capturing the provenance of a generated PDF file over an extended period of time and through a series of system and file modifications. Readers will become familiar with the ontological representation of the parsed provenance information, how the various components of the presented solution provide functionality critical to its utility, and how they can interact with this information through the use of the Protégé plugins OntoGraf and SPARQL Query.

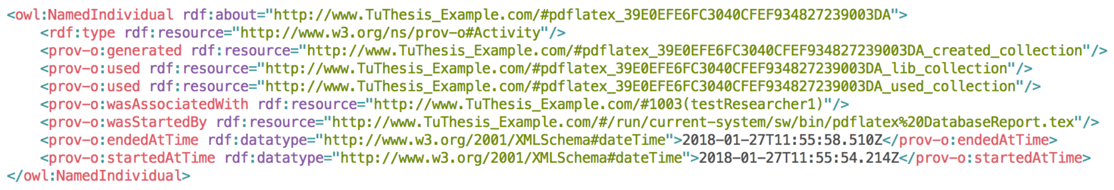
Both working examples will utilize the same optimizations with respect to the auditd rules, the excluded processes and directories, and the symlinks to be resolved. These optimizations can be seen in Table 4.1 and Figure 4.2.

4.4.1 Single Process Invocation

In Figure 4.7 we see an entry in the RDF/XML file representing the "Process_<MD5>" box in Figure 4.6. This entry represents a central point in the ontological representation of the provenance for our pdflatex working example, and is of type "prov:Activity". We can see that in this case, the MD5 is 39E0EFE6FC3040CFEF9348 27239003DA, allowing us to uniquely identify this execution of the pdflatex application from others. This entry also relates the activity to the responsible user (testResearcher1), as well as its utilized and generated/created files and resources. Finally, we see the starting and ending times of this execution.

Ignored Processes	Ignored Directories	Resolved Symlinks
dbus	/proc/	/etc/fonts/
baloo	/tmp/	/run/current-system/
baloo_file	/dev/	
baloo_file_extr	/sys/	
plasmashell		
dbus-daemon		
ls		
konsole		
dolphin		
ps		
bash		

Table 4.1: Working Example Optimizations



```
<owl:NamedIndividual rdf:about="http://www.TuThesis_Example.com/#pdflatex_39E0EFE6FC3040CFEF934827239003DA">
    <rdf:type rdf:resource="http://www.w3.org/ns/prov-o#Activity"/>
    <prov-o:generated rdf:resource="http://www.TuThesis_Example.com/#pdflatex_39E0EFE6FC3040CFEF934827239003DA_created_collection"/>
    <prov-o:used rdf:resource="http://www.TuThesis_Example.com/#pdflatex_39E0EFE6FC3040CFEF934827239003DA_lib_collection"/>
    <prov-o:used rdf:resource="http://www.TuThesis_Example.com/#pdflatex_39E0EFE6FC3040CFEF934827239003DA_used_collection"/>
    <prov-o:wasAssociatedWith rdf:resource="http://www.TuThesis_Example.com/#1003(testResearcher1)"/>
    <prov-o:wasStartedBy rdf:resource="http://www.TuThesis_Example.com/#/run/current-system/sw/bin/pdflatex%20DatabaseReport.tex"/>
    <prov-o:endedAtTime rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">2018-01-27T11:55:58.510Z</prov-o:endedAtTime>
    <prov-o:startedAtTime rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">2018-01-27T11:55:54.214Z</prov-o:startedAtTime>
</owl:NamedIndividual>
```

Figure 4.7: Example of Pdflatex Process in RDF Format

Figures 4.8, 4.9, and 4.10 depict the resources that were either created, used libraries, or other resources that are not libraries respectively. Each is type "*prov:Collection*", a subclass of "prov:Entity". The members of each collection are related to their respective entity though the "*prov:hadMember*" relationship. We see that in Figure 4.9, the collection of used libraries, the entity name for each member is the full path of the directory in which the utilized library resides. As previously discussed, this path, or more specifically the unique 160-bit, allows for the precise identification of the library dependency used at the time of execution. Entity members that belong to the remaining two collections (used, created) are represented only by the file name local to the directory in which they reside, in combination with the SHA-1 hash unique to the resources utilized version.

There are a few important distinctions to be made about how these entity members, depicted in Figure 4.11 and Figure 4.12, are represented. We see that the entity in 4.11 contains a "*prov:generatedAtTime*" data property. This time correlates directly to the "pdflatex_39E0EFE6FC3040CFEF934827239003DA_created_collection" entity, indicating the time at with the file was created. Additionally, this entity also contains a "prov:location" data property which specifies the location, or directory, of the entity. Finally, for entity members that are part of the "pdflatex_39E0EFE6FC3040CFEF934827239003DA_used _collection", there is an additional object property annotation that indicates at what time the entity was used by the *pdflatex_39E0EFE6FC3040CFEF934827239003DA* process. This can be seen in Figure 4.12, where an annotation source (pdflatex_39E0EFE6FC3040C

FEF934827239003DA__used__collection), property (*prov:hadMember*), target (DatabaseReport.acr__14dc4ed), and time (*prov:atTime*) are specified.

```
<owl:NamedIndividual rdf:about="http://www.TuThesis_Example.com/#pdflatex_39E0EFE6FC3040CFEF934827239003DA_created_collection">
    <rdf:type rdf:resource="http://www.w3.org/ns/prov-o#Collection"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.acn_4ecef95"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.aux_4ecef95"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.glo_4ecef95"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.glsdefs_4ecef95"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.idx_4ecef95"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.ist_4ecef95"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.lof_4ecef95"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.log_4ecef95"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.out_4ecef95"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.pdf_4ecef95"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.toc_4ecef95"/>
</owl:NamedIndividual>
```

Figure 4.8: Example of Pdflatex Created Collection in RDF Format

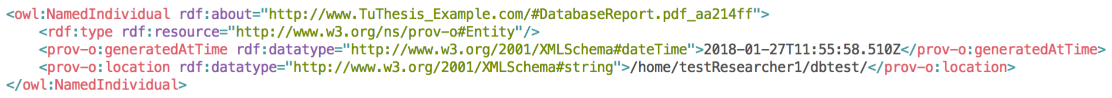
```
<owl:NamedIndividual rdf:about="http://www.TuThesis_Example.com/#pdflatex_39E0EFE6FC3040CFEF934827239003DA_lib_collection">
    <rdf:type rdf:resource="http://www.w3.org/ns/prov-o#Collection"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/0nvlpdjlw4kx34bk7i2z1asdwcv8s995-libpng-apng-1.6.31/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/202v4iam4xfv21sypr6ia6i9cz9kqs2z-libjpeg-turbo-1.5.2/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/60kv79zm1bx5amgwwi4sgg65majxhbbh-freetype-2.7.1/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/8sfng2bz5jipbs5jlw1wsx13qrh1f6h3-poppler-min-0.56.0/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/d7vzdcnscqppm2blb2b0kmaynhns0wmz-texlive-bin-2016/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/iwilqqhq14v74k3x9w3y19xwq5dcz0vx-nixos-system-nixos-17.09.1756.c99239bca0/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/iyqprhf7j4q0g6ds9ml436fg94inyx9g-lcms2-2.8/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/mvc75dp5wxslic9yrpllb8q5xlfm8ajs-bzip2-1.0.6.0.1/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/nl5m8xh30wfkniifj0sxk73nladw1j4q-openjpeg-2.1.2/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/qwxcpfxj9nz6blxnsy8k527w9n6sniqg-xz-5.2.3/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/rj8nnlz5w3aa8rpcfb5fa7km29mc3lr4-fontconfig-2.12.1-lib/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/sm0rz4i6m4f4kzvln9gkldxj7gyigwg7-libtiff-4.0.8/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/snkwsn9lijx9sq2n71b41ggrhhplgg3f-zlib-1.2.11/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/xzx1bv1d7z4mgg6sg6ly0jx609qvka4x-glibc-2.25-49/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/y5ac95kk3nb52si8zcyznjrfb45720hk-gcc-6.4.0-lib/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/z22h9k69m23rq90mvclqdi8qancw9n5z-expat-2.2.4/"/>
</owl:NamedIndividual>
```

Figure 4.9: Example of Pdflatex Library Collection in RDF Format

```
<owl:NamedIndividual rdf:about="http://www.TuThesis_Example.com/#pdflatex_39E0EFE6FC3040CFEF934827239003DA_used_collection">
    <rdf:type rdf:resource="http://www.w3.org/ns/prov-o#Collection"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.acr_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.aux_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.bbl_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.gls_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.glsdefs_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.lof_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.out_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.tex_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#DatabaseReport.toc_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#TU_INF_Logo_gray.pdf_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#TU_INF_header.pdf_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#error_2.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#legend_1.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#monomi_3.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#monomi_3_2.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#priv_db_flow_2.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#q_matrix_2.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#res_1_1.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#res_1_2.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#res_2_1.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#res_2_2.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#sdb_3.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#tpc_flow_2.png_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#vutinfth.cls_aa214ff"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/d7vzdcnscqppm2blb2b0kmaynhns0wmz-texlive-bin-2016/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/dx3h8164qrgx3gpcr3xafkayawdi5cca-poppler-data-0.4.7/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/lbcgxxkxk34s5g2dkwix16zppq7pj5wg-nixos-system-nixos-17.09.1756.c99239bca0/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/qgk5lh1rq2y9f1mz9npmkz8r7zw4g1qg-texlive-combined-full-2016/"/>
    <prov-o:hadMember rdf:resource="http://www.TuThesis_Example.com/#/nix/store/qp5fw57d38bd1n07ss4zxh88zg67c3vg-bash-4.4-p12/"/>
</owl:NamedIndividual>
```

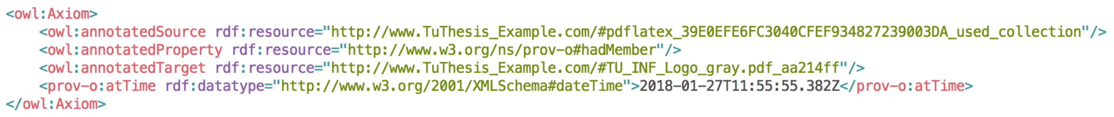
Figure 4.10: Example of Pdflatex Used Collection in RDF Format

In Figure 4.13, Figure 4.14, and Figure 4.15, we see our working example as presented by the Protégé OntoGraf plugin. As a result of the design choices in how the ontology is modeled using PROV-O, it should be apparent to the reader, through



```
<owl:NamedIndividual rdf:about="http://www.TuThesis_Example.com/#DatabaseReport.pdf_aa214ff">
    <rdf:type rdf:resource="http://www.w3.org/ns/prov-o#Entity"/>
    <prov-o:generatedAtTime rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">2018-01-27T11:55:58.510Z</prov-o:generatedAtTime>
    <prov-o:location rdf:datatype="http://www.w3.org/2001/XMLSchema#string">/home/testResearcher1/dbtest/</prov-o:location>
</owl:NamedIndividual>
```

Figure 4.11: Example of Pdflatex Entity in RDF Format



```
<owl:Axiom>
    <owl:annotatedSource rdf:resource="http://www.TuThesis_Example.com/#pdflatex_39E0EFE6FC3040CFEF934827239003DA_used_collection"/>
    <owl:annotatedProperty rdf:resource="http://www.w3.org/ns/prov-o#hadMember"/>
    <owl:annotatedTarget rdf:resource="http://www.TuThesis_Example.com/#TU_INF_Logo_gray.pdf_aa214ff"/>
    <prov-o:atTime rdf:datatype="http://www.w3.org/2001/XMLSchema#dateTime">2018-01-27T11:55:55.382Z</prov-o:atTime>
</owl:Axiom>
```

Figure 4.12: Example of Pdflatex Annotation in RDF Format

these figures, the level of control the user has regarding the granularity of the presented provenance information. Specifically, the chosen model allows the user to see only the provenance information they wish to see, should it be the utilized libraries, general resources, or creates files. Figure 4.13 shows us the "main entry" for the *pdflatex_39E** process in the ontology. That is, we see the main entities that this process is related to: the command line responsible for its invocation, the library, used, and three resource collections (prefixed by *pdflatex_39E**), and the associated user. The connections are color coded based on the type of relationship that exists between any two given ontology entities, and is viewable by mouse-over as seen in Figure 4.18.

The Protégé OntoGraf plugin also allows for the more detailed inspection of the ontology objects by mousing over a given node. We can see an example of this in Figures 4.16 and 4.17. In the displayed tooltips, all object property and data property assertions are displayed. We note here however that object property annotations are not displayed by the OntoGraf plugin. Thus, OntoGraf will not display the *xsd:dateTime* object property annotation indicating when a resource was utilized by a process. Additionally, the relationships between nodes can also be viewed by mousing over the arcs connecting the ontology nodes, as seen in Figure 4.18.

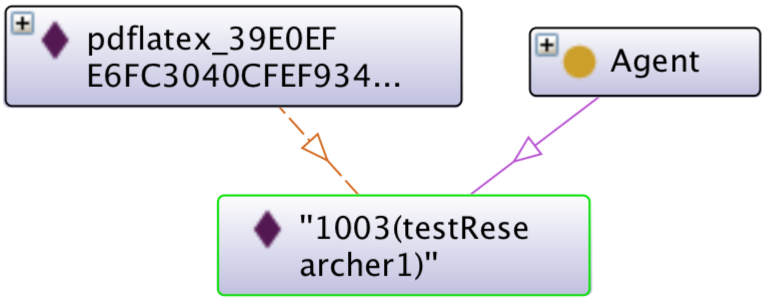


Figure 4.13: Example of Pdflatex User in Protégé OntoGraf

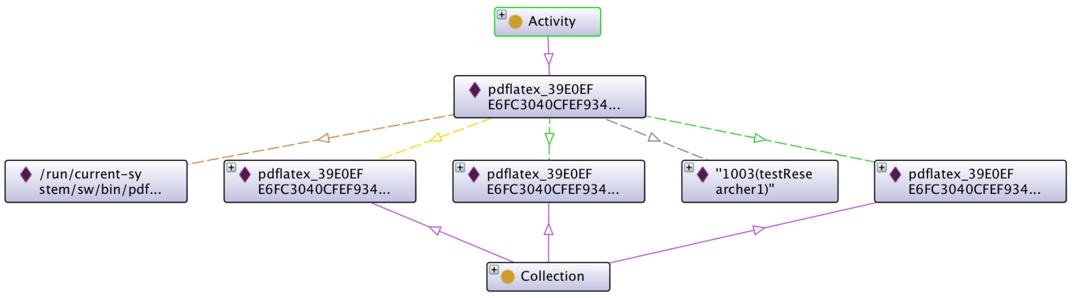

Activity
pdflatex_39E0EF E6FC3040CFEF934...
/run/current-sy stem/sw/bin/pdf...
pdflatex_39E0EF E6FC3040CFEF934...
pdflatex_39E0EF E6FC3040CFEF934...
"1003(testRese archer1)"
pdflatex_39E0EF E6FC3040CFEF934...
Collection

Figure 4.14: Example of Pdflatex Process Activity in Protégé OntoGraf

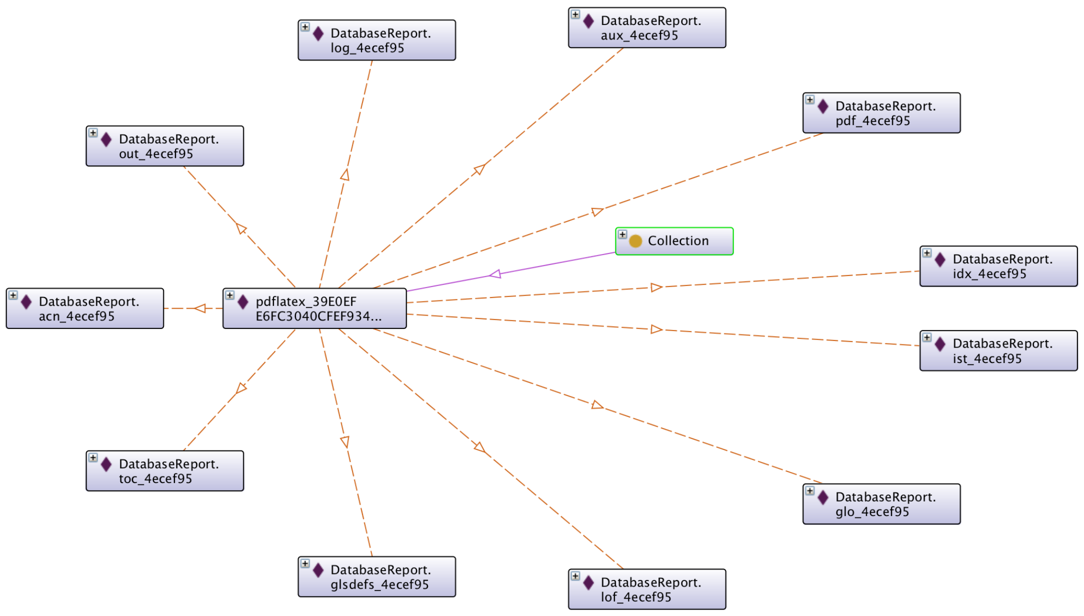
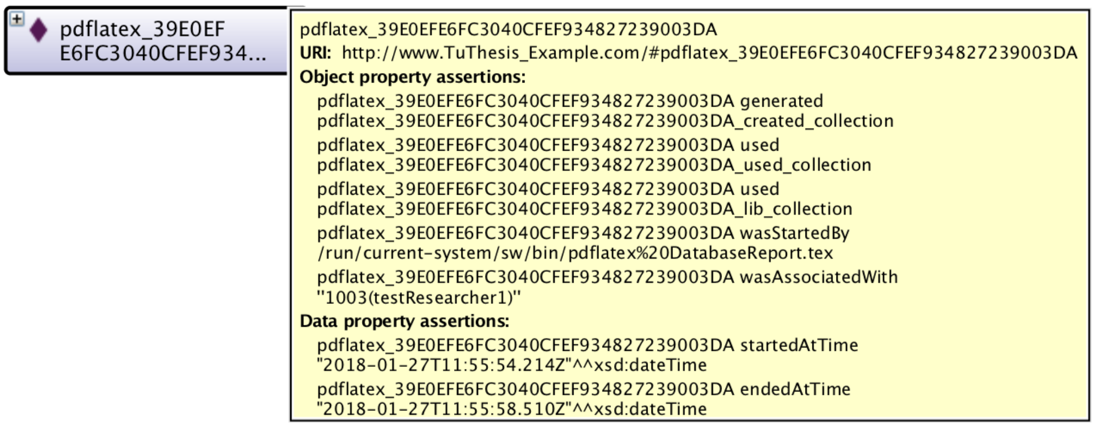

DatabaseReport. log_4ecef95
DatabaseReport. aux_4ecef95
DatabaseReport. pdf_4ecef95
DatabaseReport. out_4ecef95
Collection
DatabaseReport. idx_4ecef95
DatabaseReport. acn_4ecef95
pdflatex_39E0EF E6FC3040CFEF934...
DatabaseReport. ist_4ecef95
DatabaseReport. toc_4ecef95
DatabaseReport. glo_4ecef95
DatabaseReport. glsdefs_4ecef95
DatabaseReport. lof_4ecef95

Figure 4.15: Example of Pdflatex Used Collection in Protégé OntoGraf

Finally, in Figures 4.19, 4.20, and 4.21, we see how users can query the ontology in a very specific manner. We provide examples for how a user can query for all processes with which a user was associated, all processes that have used a specific library, and all files that were utilized at a designated time. These SPARQL queries allow for the users to find precisely the information they are searching for.

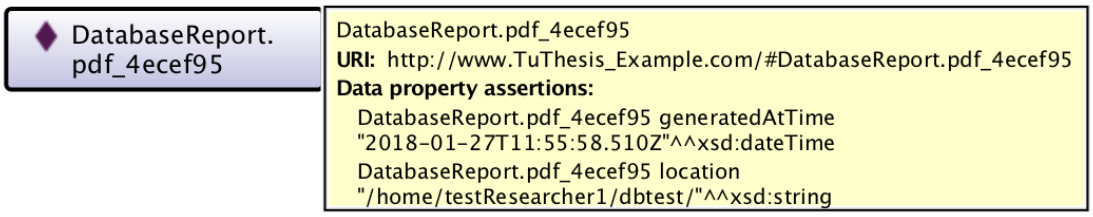
4.4.2 Ontology Over Time

To explore the utility of the various components of the presented solution, in this section we present a variety of examples that reveal its capability and application in real world scenarios.



pdflatex_39E0EFE6FC3040CFEF934827239003DA
URI: http://www.TuThesis_Example.com/#pdflatex_39E0EFE6FC3040CFEF934827239003DA
Object property assertions:
pdflatex_39E0EFE6FC3040CFEF934827239003DA generated pdflatex_39E0EFE6FC3040CFEF934827239003DA_created_collection
pdflatex_39E0EFE6FC3040CFEF934827239003DA used pdflatex_39E0EFE6FC3040CFEF934827239003DA_used_collection
pdflatex_39E0EFE6FC3040CFEF934827239003DA used pdflatex_39E0EFE6FC3040CFEF934827239003DA_lib_collection
pdflatex_39E0EFE6FC3040CFEF934827239003DA wasStartedBy /run/current-system/sw/bin/pdflatex%20DatabaseReport.tex
pdflatex_39E0EFE6FC3040CFEF934827239003DA wasAssociatedWith "1003(testResearcher1)"
Data property assertions:
pdflatex_39E0EFE6FC3040CFEF934827239003DA startedAtTime "2018-01-27T11:55:54.214Z"^^xsd:dateTime
pdflatex_39E0EFE6FC3040CFEF934827239003DA endedAtTime "2018-01-27T11:55:58.510Z"^^xsd:dateTime

Figure 4.16: Example of Pdflatex Process Tooltip



DatabaseReport.pdf_4ecef95
URI: http://www.TuThesis_Example.com/#DatabaseReport.pdf_4ecef95
Data property assertions:
DatabaseReport.pdf_4ecef95 generatedAtTime "2018-01-27T11:55:58.510Z"^^xsd:dateTime
DatabaseReport.pdf_4ecef95 location "/home/testResearcher1/dbtest/"^^xsd:string

Figure 4.17: Example of Pdflatex Entity Tooltip

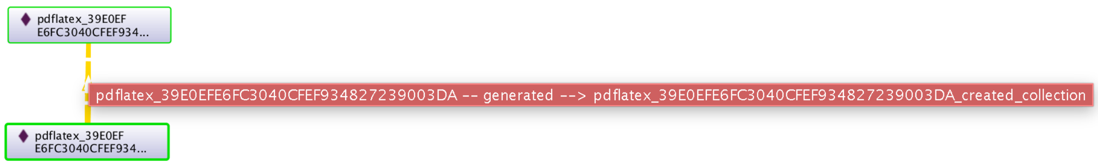
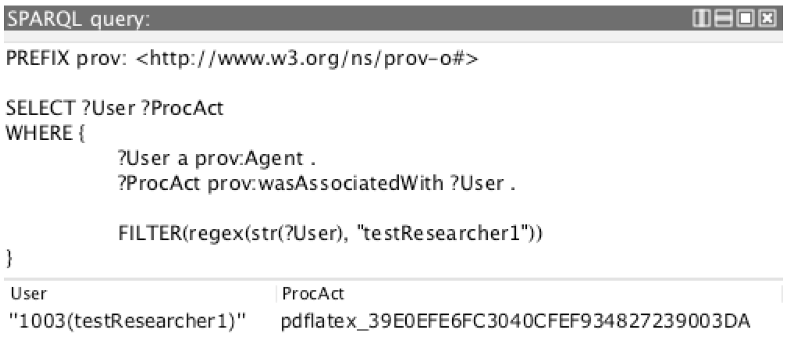

pdflatex_39E0EF
E6FC3040CFEF934...
pdflatex_39E0EFE6FC3040CFEF934827239003DA -- generated --> pdflatex_39E0EFE6FC3040CFEF934827239003DA_created_collection
pdflatex_39E0EF
E6FC3040CFEF934...

Figure 4.18: Example of Pdflatex Relationship Label

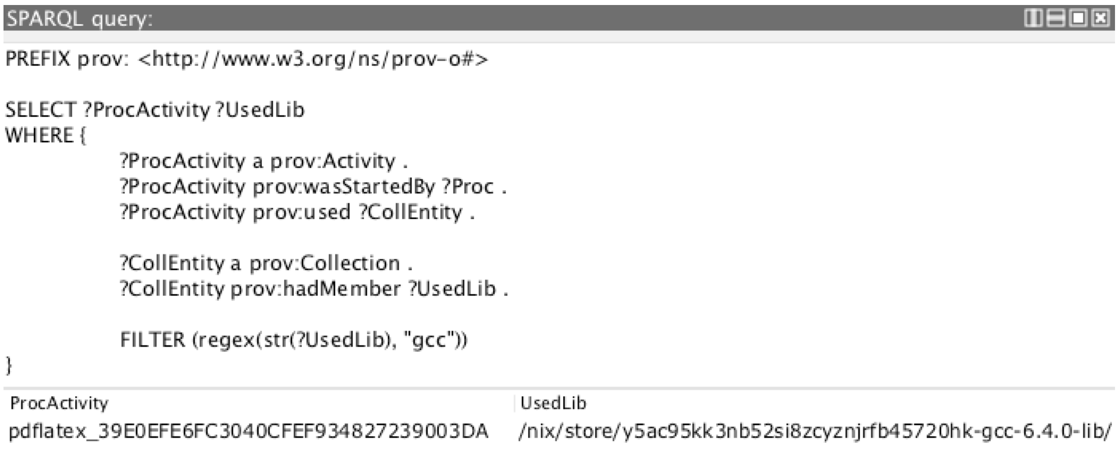
As is the case for much of the computational research being performed today, the data on which the research will be performed is obtained from from an external database or repository on the internet. In our first example, we demonstrate our solution's ability to track the provenance of data starting from its time of acquisition and continuing through various stages of its application on the solution system. Specifically, we will download a TAR archive containing a revision of this very LaTeX document and all of its resources, extract the contents of the TAR archive to a directory in our test user's home directory, and compile the LaTeX document to obtain a PDF file.

We begin first by looking at the ontology resulting from the process described in Figure 4.22. In this figure there are a number of important observations to be made.



```
SPARQL query:
PREFIX prov: <http://www.w3.org/ns/prov-o#>

SELECT ?User ?ProcAct
WHERE {
		?User a prov:Agent .
		?ProcAct prov:wasAssociatedWith ?User .

		FILTER(regex(str(?User), "testResearcher1"))
}

User                       ProcAct
"1003(testResearcher1)"    pdflatex_39E0EFE6FC3040CFEF934827239003DA
```

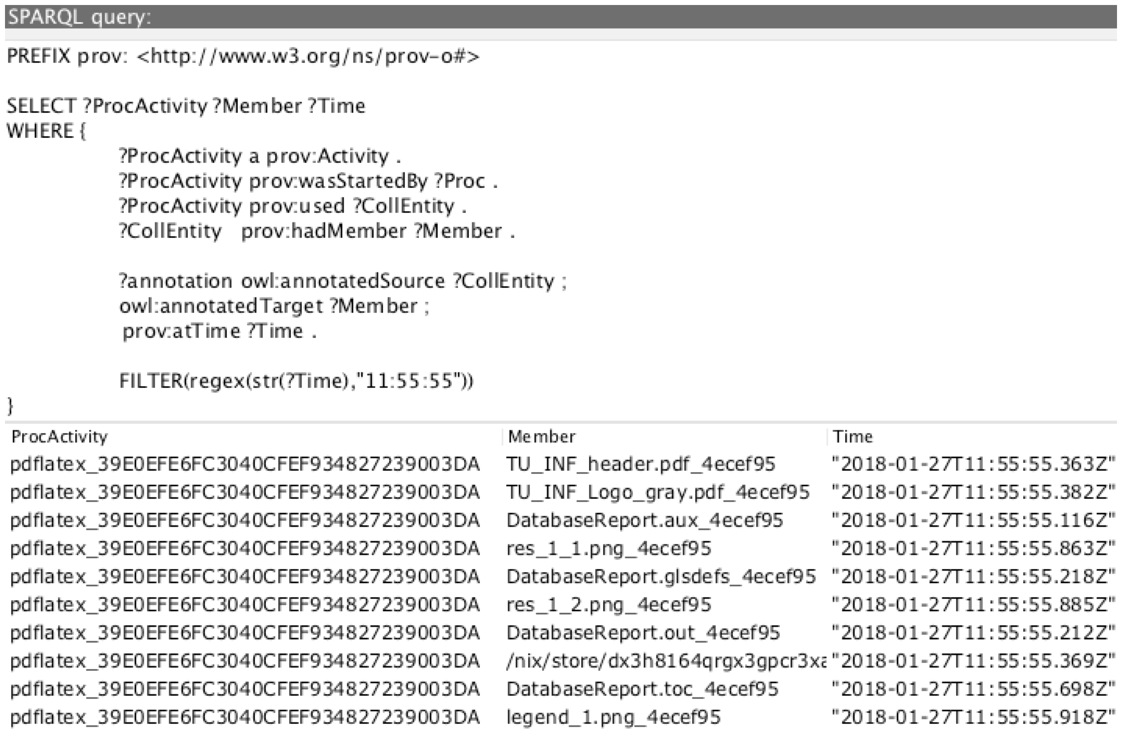
Figure 4.19: Example of a SPARQL query selecting all processes associated with a specific user



```
SPARQL query:
PREFIX prov: <http://www.w3.org/ns/prov-o#>

SELECT ?ProcActivity ?UsedLib
WHERE {
		?ProcActivity a prov:Activity .
		?ProcActivity prov:wasStartedBy ?Proc .
		?ProcActivity prov:used ?CollEntity .

		?CollEntity a prov:Collection .
		?CollEntity prov:hadMember ?UsedLib .

		FILTER (regex(str(?UsedLib), "gcc"))
}

ProcActivity                                  UsedLib
pdflatex_39E0EFE6FC3040CFEF934827239003DA     /nix/store/y5ac95kk3nb52si8zcyznjrfb45720hk-gcc-6.4.0-lib/
```

Figure 4.20: Example of a SPARQL query selecting all processes using a specific library

There are three applications that interact with the data represented in this ontology: wget, tar, and pdflatex. Each application is associated with its corresponding process, which is in turn associated with the resource collections as defined in previous section. We are able to observe that the *wget_7DD** process was used to obtain the *THESIS_LATEX.tar* archive file. This archive was subsequently unpacked by the *tar_2EB** process, and the resulting files used by the *pdflatex_569** process. The three entities in the middle of the image are the *_used* collections (marked [1] and [2]) of the *tar_2EB** and *pdflatex_569** processes, and the *_created* collection (marked [3]) of the *pdflatex_569** process. We note that the two *_used* collections are related to, and have thus utilized, many of the same resources. Additionally, we observe that *pdflatex_569** has created new revisions of many of these files, including of course "RRoth_Thesis.pdf_3da3617" (marked [4]). As the process prefixes do not allow one to see the entire name of the entity without mousing over it, Figure 4.22 has been marked to indicate the entity types as follows: [M] indicates the main entries of a process, and [C] indicates the collection entities that a given process is related to.



SPARQL query:

```
PREFIX prov: <http://www.w3.org/ns/prov-o#>

SELECT ?ProcActivity ?Member ?Time
WHERE {
            ?ProcActivity a prov:Activity .
            ?ProcActivity prov:wasStartedBy ?Proc .
            ?ProcActivity prov:used ?CollEntity .
            ?CollEntity   prov:hadMember ?Member .

            ?annotation owl:annotatedSource ?CollEntity ;
            owl:annotatedTarget ?Member ;
             prov:atTime ?Time .

            FILTER(regex(str(?Time),"11:55:55"))
}
```

```
ProcActivity                                  Member                          Time
pdflatex_39E0EFE6FC3040CFEF934827239003DA    TU_INF_header.pdf_4ecef95       "2018-01-27T11:55:55.363Z"
pdflatex_39E0EFE6FC3040CFEF934827239003DA    TU_INF_Logo_gray.pdf_4ecef95    "2018-01-27T11:55:55.382Z"
pdflatex_39E0EFE6FC3040CFEF934827239003DA    DatabaseReport.aux_4ecef95      "2018-01-27T11:55:55.116Z"
pdflatex_39E0EFE6FC3040CFEF934827239003DA    res_1_1.png_4ecef95             "2018-01-27T11:55:55.863Z"
pdflatex_39E0EFE6FC3040CFEF934827239003DA    DatabaseReport.glsdefs_4ecef95  "2018-01-27T11:55:55.218Z"
pdflatex_39E0EFE6FC3040CFEF934827239003DA    res_1_2.png_4ecef95             "2018-01-27T11:55:55.885Z"
pdflatex_39E0EFE6FC3040CFEF934827239003DA    DatabaseReport.out_4ecef95      "2018-01-27T11:55:55.212Z"
pdflatex_39E0EFE6FC3040CFEF934827239003DA    /nix/store/dx3h8164qrgx3gpcr3xa "2018-01-27T11:55:55.369Z"
pdflatex_39E0EFE6FC3040CFEF934827239003DA    DatabaseReport.toc_4ecef95      "2018-01-27T11:55:55.698Z"
pdflatex_39E0EFE6FC3040CFEF934827239003DA    legend_1.png_4ecef95            "2018-01-27T11:55:55.918Z"
```

Figure 4.21: Example of a SPARQL query selecting entities used at a specific time

In Figure 4.23 we highlight the *wget_7DD** process to demonstrate the ability to capture the command-line used for its execution. Importantly, we observe that this information includes the Uniform Resource Locator (URL) from which the file was obtained. We note that the *prov:wasStartedBy* object property is encoded by the *rdf:resource* XML attribute which enforces white space normalization. Therefore, spaces in this object property are represented by *%20*.

To further investigate the provenance of the downloaded TAR archive and resulting data, we demonstrate a SPARQL query in Figure 4.24. In this SPARQL query, we query for *_collection* entities containing the resources "RRoth_Thesis.pdf", "THESIS_LATEX.tar", and the class file used to generate the PDF file, "vutinfth.cls". The query results allow users to more easily follow the provenance of some of the critical resources in this working example. Specifically, the results make it clear that *tar_2EB** used the exact TAR archive obtained by *wget_7DD**, and that *pdflatex_569** used the exact class file extracted from the TAR archive by *tar_2EB**. Furthermore, we observe that *pdflatex_569** also created a new version of the thesis PDF, "RRoth_Thesis.pdf_3da3617".

This brings us to the next step in this working example of the solution system; demonstrating the Git Version Control and NixOS functionality. To do so, we invoke multiple executions of the pdflatex executable to generate this thesis' PDF file. More specifically, each execution of pdflatex will utilize different resources and even different (versions of) libraries.

In Figure 4.25 we see the OntoGraf representation of the ontology resulting from our multiple pdflatex invocations. The entity in the center of the image represents the command-line used to invoke both pdflatex processes. Additionally, we see each process' created files. We note that some files will not always change between executions, and are thus re-used across multiple process executions. Should a file change from one execution to the next, depending on how the process utilized this new version, it will be included in the *_created* collection, and possibly the *_used* collection, of the new process execution, and will be suffixed with a new SHA-1 hash. This behavior is also exhibited through multiple executions of the pdflatex application, and often times includes files such as generated bibliography or acronym files. In Figure 4.25, we see that the glossary files (marked with [1]), the index file (marked with [2]), and the list of tables file (marked with [3]), did not change from one execution of pdflatex to the next, and thus both processes have created identical versions of these files. However, we observe that files such as logs, and of course the generated PDF file, have been rebuilt, changed, and have thus received new SHA-1 hashes from Git. These SHA-1 hashes can be used in Git utility commands to obtain the exact version utilized, as described by the ontology.

In Figure 4.26 we see the results of a SPARQL query for *_collection* entities containing our generated PDF and our newly added images as see in Figure 4.22 and Figure 4.24. The most important observations to be made from this figure are that (1) there are three versions of the PDF generated in this working example, that (2) only a single version of this PDF (_a2816ef) contains the two image resources (Figure 4.22 as *sol_prot_working_download_to_pdf_f77298b* and Figure 4.24 as *sol_sparql_working_download_to_pdf_f77298b*), and that (3) each PDF created by the pdflatex process utilized a different version of the LaTeX file. Changes between the various versions of a given file, for example the three versions of the LaTeX file seen in Figure 4.26, can be visualized through diff applications such as Meld or BeyondCompare, or with the Git repository browser, Gitk. Additionally, we observe that the *pdflatex_569** process (the first invocation of pdflatex in our ontology) uses the exact version of the LaTeX document obtained when unpacking the TAR archive. This thus demonstrates the ability of the solution system to allow users to precisely identify which resources were used by a given process, and subsequently in what files resulting from the execution of said process these resources were used.

In additional to capturing the provenance of our working example across edits, we have also done so across NixOS updates. Specifically, between pdflatex executions, we have updated the entire NixOS to a newer version; from 17.09.1756.c99239bca0 to 17.09.2905.c1d9aff56e0. This process may sound like a non-trivial task, however due the nature of the NixOS and its reliance on the Nix Package Manager, this update is as simple as switching channels and rebuilding the NixOS. In this particular case, the rebuilding of the NixOS did not require a restart of the system. Below are the commands utilized to perform this task.

```
nix-channel –add https://nixos.org/channels/channel-name nixos
nixos-rebuild switch –upgrade
```

In Figure 4.27 we see the results of a SPARQL query for libraries used by two executions of the pdflatex application. Specifically, we query only for libraries that are not members of more than one *lib_collection* entity to demonstrate the different libraries that were used by the pdflatex processes. Additionally, we limit the results of this query to a select few libraries (gcc, nixos, fontconfig, and glibc) to maintain the effectiveness of the working example. We first observe that each pdflatex process was executed on a different version of the NixOS. These high level "nixos" directories are the resolved symlink paths of the */run/current-system/* directory, in which many system specific libraries are located. Second, we observe that both pdflatex processes utilized the "gcc-6.4.0" library. However, the respective members of each processes' *lib_collection* entity indicate via the 160-bit hash that these two libraries are not 100% identical. This is a consequence of the libraries being compiled against different dependencies, and with perhaps different configuration options as a result the environment in which they were compiled (eg: different NixOS versions). We notice that this is also the case for the "fontconfig-2.12.1" library. Importantly, as mentioned already in this section and in section 3.1.2, this information is crucial in providing users with the ability to identify exact resources, and of course for the concept of reproducibility as a whole. Finally, we observe that each pdflatex process utilized entirely different versions of the "glibc" library, with the "old" NixOS version using version 2.25-49, and the updated NixOS version 2.25-123.

4.4.3 Performance and Complexity

In this section we look to explore the performance and complexity of the presented solution system. In doing so, we will demonstrate the effectiveness of our design decisions in providing a feasible solution that contributes to the increased reproducibility of computational research.

To better understand the presented solutions performance, we will compare the log files and resulting ontologies for the executions of two largely different process executions: pdflatex and Firefox. Pdflatex will be used to compile a PDF file from a LaTeX document, and Firefox will be used to visit a website and download a single file to testResearcher1's home directory. These two processes will aid in exposing the complexities of both command-line and GUI applications and the challenges of dealing with the operations each respective process invokes on the solution system. As detailed in sections 4.3.2 and 4.3.3, there are a number of configurable options available to the users that allow for the fine-tuning of ontolgies and the log files from which they were created. These options include monitoring specific syscalls with auditd, excluding entry types and the actions of specific UIDs with auditd, exclusions lists for processes and directories, and a white-list for symlink resolution.

In tables 4.2 and 4.3 we present the results of our two process executions by breaking up the executions into three sections. The top most section presents the results of the process executions using the optimal configuration as detailed in Table 4.1 and Figure 4.2. The middle section presents the results of the process executions after having removed the rule(s) corresponding to the row title from the auditd rules, while maintaining the optimal Auditd Log Parser configuration. The final section presents the results of the process executions after having removed the argument corresponding to the row title from the execution of the Auditd Log Parser, but while still maintaining the optimal auditd rules. That is, respective to each row in tables 4.2 and 4.3, the Auditd Log Parser will not ignore specified processes or directories, and will not resolve the specified symlinks to their Nix Store directories. As the optimizations removed in this final section do not have an impact on the generated log file, the Auditd Log Parser is executed on the log file generated during the process execution utilizing the optimal configuration. We note that the "No Auditd Exclusion Rules" execution requires special attention, which will be addressed later in this section.

The columns of tables 4.2 and 4.3 are defined as follows: size is defined as the number of lines the respective logs and ontologies contain, execution time is defined as the difference between the first and last time stamps of the generated log files, and processing time is defined as the time required for the Auditd Log Parser to parse the respective log files into the resulting ontology.

	Log Size	Ontology Size	Execution Time	Processing Time
Optimal Configuration	5,057	1,551	12.3s	5.2s
Monitoring "write" Syscalls	8,080	1,692	14.1s	6.2s
No Auditd Exclusion Rules*	129,574	5,792	42.7s	22.8s
No Process Exclusions	5,057	3,086	12.3s	7.5s
No Directory Exclusions	5,057	1,567	12.3s	5.3s
No Symlink Resolving	5,057	1,567	12.3s	5.4s

Table 4.2: Pdflatex Execution Results

In both tables 4.2 and 4.3 we observe that our optimal configuration has produced the best results for both process executions. That is, both executions yielded the smallest logs (5,057 and 13,870 lines), the smallest ontology (1,551 and 1,449 lines), the shortest execution times (12.3 and 31.9 seconds), and the shortest processing times (5.2 and 1.3 seconds). Interesting is the discrepancy between the log sizes and processing times between the pdflatex command-line application and Firefox GUI application. Although the pdflatex log size is less than one-half the size of that for Firefox, the processing time is almost four-times as long. This discrepancy is a result of the Git Version Control component of our solution system. Specifically, the pdflatex application creates 12 files

	Log Size	Ontology Size	Execution Time	Processing Time
Optimal Configuration	13,870	1,449	31.9s	1.3s
Monitoring "write" Syscalls	29,960	1,663	40.1s	2.3s
No Auditd Exclusion Rules*	158,480	8,954	55.3s	23.0s
No Process Exclusions	13,870	2,435	31.9s	2.4s
No Directory Exclusions	13,870	2,031	31.9s	1.3s
No Symlink Resolving	13,870	11,978	31.9s	1.6s

Table 4.3: Firefox Execution Results

in testResearcher1's home directory, while Firefox only creates the single downloaded file. As new files in our test user's home directory are detected, the Auditd Log Parser must perform the expensive operation of retrieving the current status of the Git repository and determining if the files must be committed.

We next note the difference in log size between executions with and without auditd monitoring "write" syscalls. Log sizes inflated in both applications, with the log for pdflatex containing 59.78% more lines, and the log for Firefox containing 116.01% more lines. This results in a one second longer processing time for each application's log file. We also observe that the execution times of each respective application have increased. As auditd must now log more audit entries, it requires more computing power to do so, thus decreasing the resources available to the application(s) running on the solution system and ultimately slowing down the system.

The removal of the exclusion rules from auditd resulted in the largest difference in the viability of the solution system. So much so in fact, that a change to the auditd configuration was required in order to provide the results accurately. Without the exclusion rules, specifically the UID exclusion rules, the audit entries made to the log file(s) arrived at a volume too high to process in time. That is, due to our configuration dictating that the log files should rotate when they reach 5MB, rotating occurred before the Auditd Log Parser had successfully parsed the entire log. This resulted in provenance information being lost and not represented in the ontology when exported. To combat this issue, the configuration was changed to allow for each log file to grow to a maximum of 20MB before rotating. This increased size restriction allowed for the Auditd Log Parser to successfully parse the total provenance information resulting from the execution of the two applications. However, this demonstrates a limit in regard to the Auditd Log Parser's efficiency, and the importance of optimizing its functionality in specific critical areas.

Both execution and processing times were also at their highest during the executions without the exclusion rules. One may be inclined to conclude that because the processing times of each respective application are well under the indicated execution times, that the Auditd Log Parser should have been able to easily capture all of the process execution's provenance information. However, we must take into account the fact that entries made to the log files are not consistent in their rate, and fluctuate with respect to the processes currently being executed on the system. In the case of the executions without the auditd exclusion rules, this results in both the Auditd Log Parser process and all spawned child processes writing auditd entries to the log file.

We now turn our attention to the final section of tables 4.2 and 4.3 where the effects of the Auditd Log Parser optimizations can be observed. In all cases, the removal of a single optimization resulted in a less efficient implementation, with both processing times and ontology sizes increasing when compared to the optimal configuration. However, noteworthy is the impact that resolving the symlinks had on each application's execution. In particular, the ontology size for the Firefox execution increased by 726.64% compared to that of the optimal configuration, while the pdflatex execution increased by only 1.03%. This highlights a fundamental difference in the computational work performed by command-line applications and GUI applications, and it should be of no surprise that GUI applications require additional resources in terms of fonts, graphics, libraries, and frameworks. Our optimal configuration allowed for all resources contained within specific (Nix Store) directories (see Table 4.1) to be represented in the ontology by only the top level directory in which they are contained. Removing this functionality, in the case of Firefox, results in thousands of font and graphic resources being individually referenced in the ontology.

Also noteworthy is how the removal of the process exclusion argument affected the pdflatex application. As this is a command-line application executed in the bash shell, bash appears often in the log file(s) and thus in the resulting ontology. As bash is only used to invoke the pdflatex application and has no affect on the data being manipulated by the pdflatex application itself, it it safe to exclude it from the ontology.

4.5 Summary

In this chapter we presented the details of our proposed solution system. We have covered how the resources, technologies, components, and concepts covered in Chapter 3 have been employed to assists researchers in tracking the provenance of data on the host system. Additionally, have seen how the NixOS (section 4.3.1) acts as the backbone for our proposed solution, upon which core components such as the Linux Audit Daemon (section 4.3.2) and the Auditd Log Parser (section 4.3.3) are implemented. Additionally, we demonstrated how the Git Version Control Component allows for the precise identification of resources both used and created within our testResearcher1's home directory. Most importantly, the model in which the provenance data is presented in section 4.3.7, and put to use in section 4.4. Users were familiarized with the exact specifications and structure of the ontology resulting from the proposed solution system through the working examples with pdflatex, tar, and wget. Specifically, the RDF/XML representation of the ontology was presented and its features examined. Additionally, we demonstrated how the Protégé plugins OntoGraf and SPARQL Query can be used to interact with and visualize the ontology.

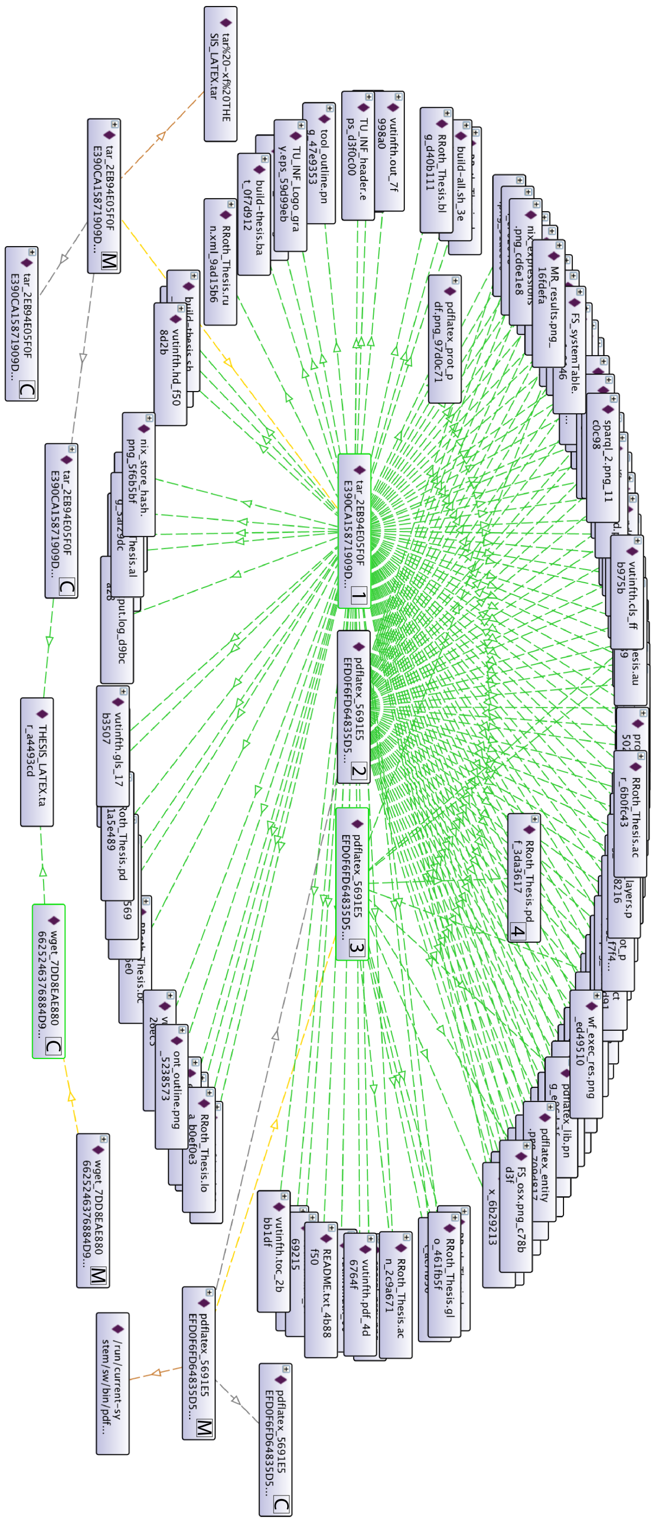
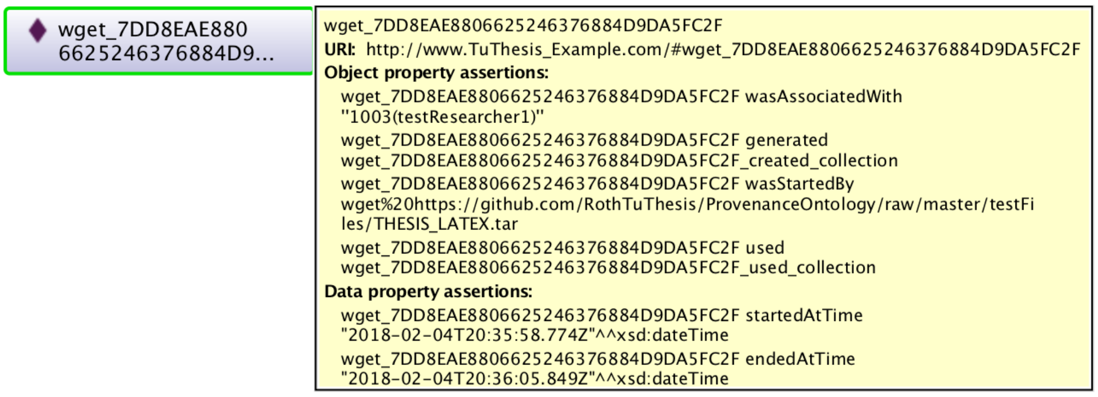
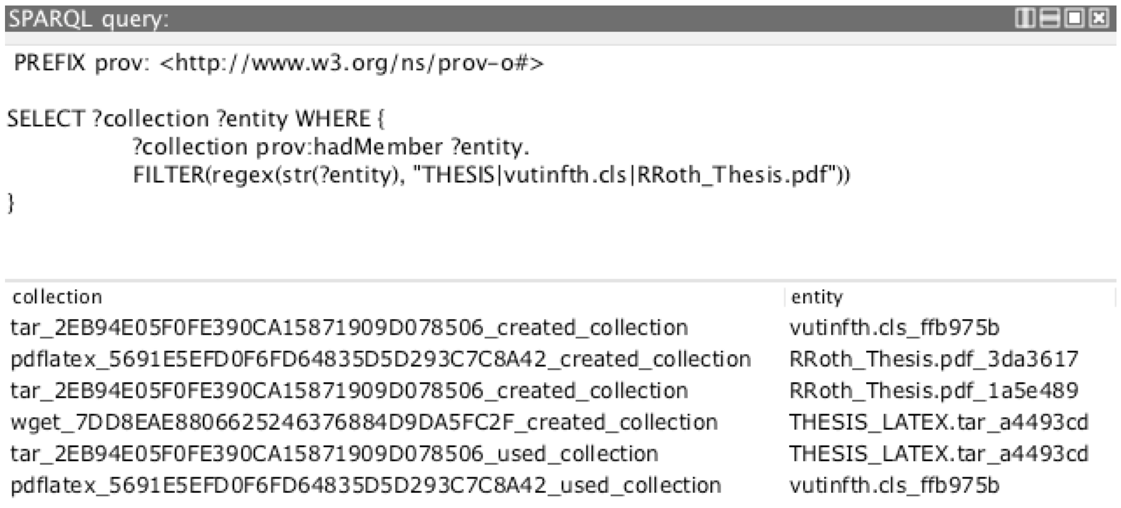

tar%20-xf%20THESIS_LATEX.tar
tar_2EB94E05F0FE390CA15871909D...
tar_2EB94E05F0FE390CA15871909D...
THESIS_LATEX.tar_a4493cd
wget_7DD8EAE8806625246376884D9...
wget_7DD8EAE8806625246376884D9...
pdflatex_5691E5EFD0F6FD64835D5...
pdflatex_5691E5EFD0F6FD64835D5...
/run/current-system/sw/bin/pdf...
RRoth_Thesis.pdf_3da3617
README.txt_4b88f50
vutinfth.toc_2bbb1df
vutinfth.pdf_4d6764f
RRoth_Thesis.acn_2c9a671
vutinfth.cls_ffb975b
sparql_2.png_11c0c98
FS_systemTable.
MR_results.png_16fdefa
nix_expressions.png_cd6e1e8
pdflatex_prot_pdf.png_97d0c71
RRoth_Thesis.blg_d40b111
vutinfth.out_7f998a0
TU_INF_header.eps_d3f0c00
tool_outline.png_47e9353
TU_INF_Logo_grey.eps_59d99eb
build-thesis.bat_07fd912
RRoth_Thesis.run.xml_9ad15b6
vutinfth.hd_f508d2b
nix_store_hash.png_5f6b5bf
vutinfth.gls_17b3507
RRoth_Thesis.lo a_b0ef0e3
wf_exec_res.png_ed49510
FS_osx.png_c78bd3f
RRoth_Thesis.glo_461fb5f

Figure 4.22: Provenance of PDF file from downloading to compiling



```
wget_7DD8EAE880
6625246376884D9...

wget_7DD8EAE8806625246376884D9DA5FC2F
URI: http://www.TuThesis_Example.com/#wget_7DD8EAE8806625246376884D9DA5FC2F
Object property assertions:
  wget_7DD8EAE8806625246376884D9DA5FC2F wasAssociatedWith
  "1003(testResearcher1)"
  wget_7DD8EAE8806625246376884D9DA5FC2F generated
  wget_7DD8EAE8806625246376884D9DA5FC2F_created_collection
  wget_7DD8EAE8806625246376884D9DA5FC2F wasStartedBy
  wget%20https://github.com/RothTuThesis/ProvenanceOntology/raw/master/testFi
  les/THESIS_LATEX.tar
  wget_7DD8EAE8806625246376884D9DA5FC2F used
  wget_7DD8EAE8806625246376884D9DA5FC2F_used_collection
Data property assertions:
  wget_7DD8EAE8806625246376884D9DA5FC2F startedAtTime
  "2018-02-04T20:35:58.774Z"^^xsd:dateTime
  wget_7DD8EAE8806625246376884D9DA5FC2F endedAtTime
  "2018-02-04T20:36:05.849Z"^^xsd:dateTime
```

Figure 4.23: Wget tooltip indicating source URL



SPARQL query:

```
PREFIX prov: <http://www.w3.org/ns/prov-o#>

SELECT ?collection ?entity WHERE {
        ?collection prov:hadMember ?entity.
        FILTER(regex(str(?entity), "THESIS|vutinfth.cls|RRoth_Thesis.pdf"))
}
```

collection	entity
tar_2EB94E05F0FE390CA15871909D078506_created_collection	vutinfth.cls_ffb975b
pdflatex_5691E5EFD0F6FD64835D5D293C7C8A42_created_collection	RRoth_Thesis.pdf_3da3617
tar_2EB94E05F0FE390CA15871909D078506_created_collection	RRoth_Thesis.pdf_1a5e489
wget_7DD8EAE8806625246376884D9DA5FC2F_created_collection	THESIS_LATEX.tar_a4493cd
tar_2EB94E05F0FE390CA15871909D078506_used_collection	THESIS_LATEX.tar_a4493cd
pdflatex_5691E5EFD0F6FD64835D5D293C7C8A42_used_collection	vutinfth.cls_ffb975b

Figure 4.24: SPARQL query demonstrating working example provenance

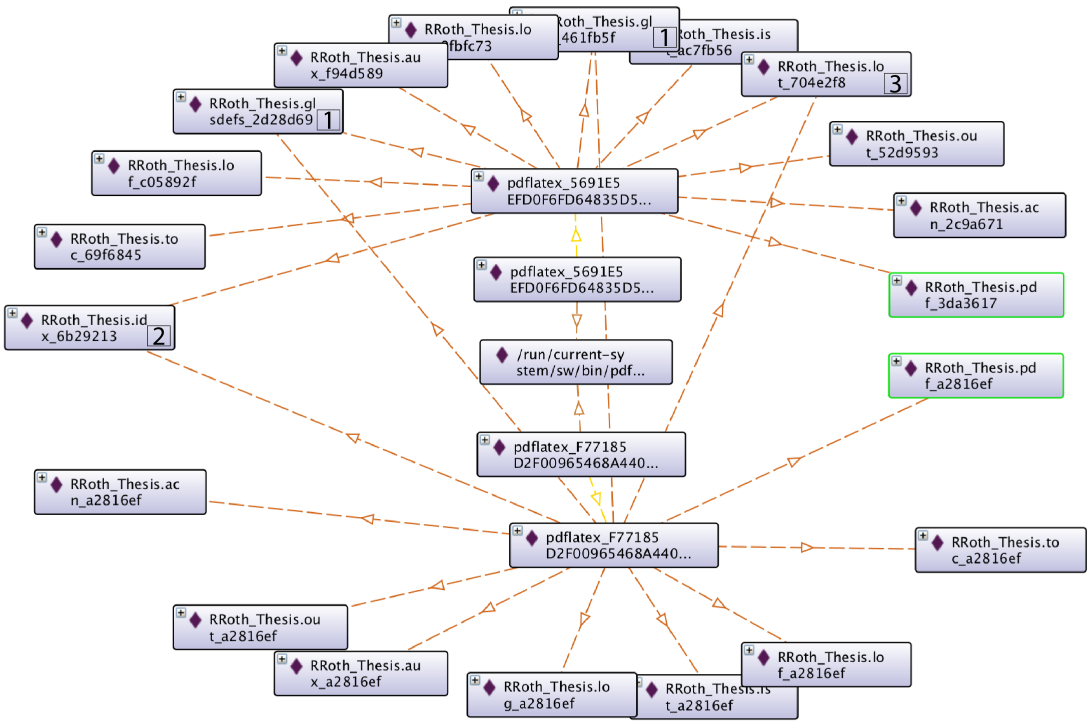
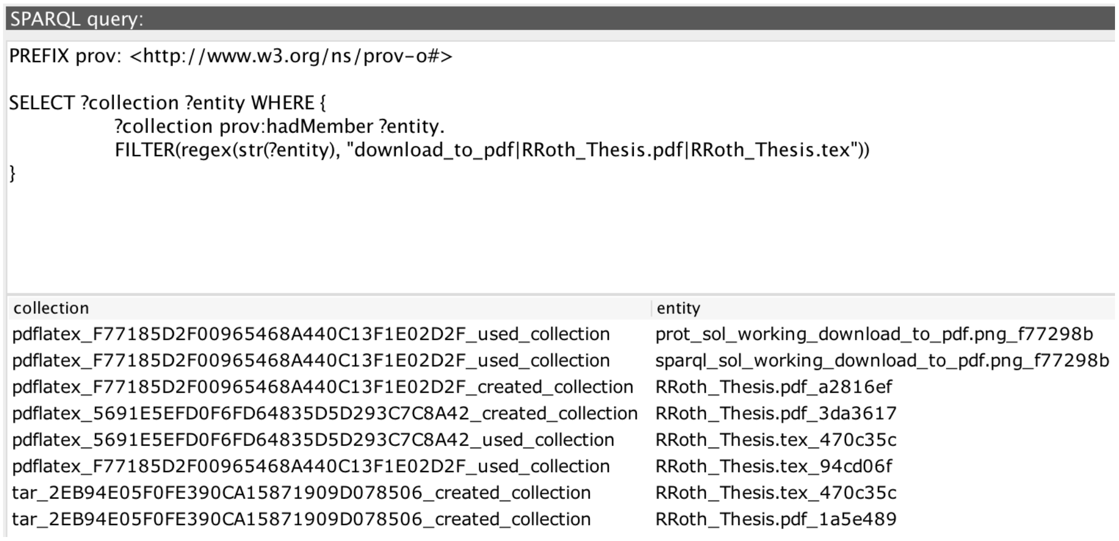


Figure 4.25: OntoGraf representation of provenance of multiple pdflatex executions

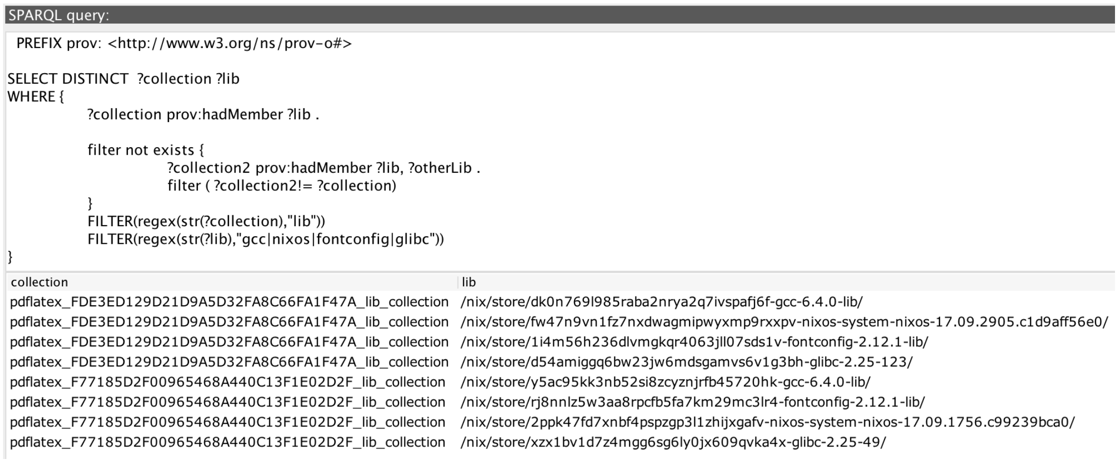


SPARQL query:

```
PREFIX prov: <http://www.w3.org/ns/prov-o#>

SELECT ?collection ?entity WHERE {
        ?collection prov:hadMember ?entity.
        FILTER(regex(str(?entity), "download_to_pdf|RRoth_Thesis.pdf|RRoth_Thesis.tex"))
}
```

collection	entity
pdflatex_F77185D2F00965468A440C13F1E02D2F_used_collection	prot_sol_working_download_to_pdf.png_f77298b
pdflatex_F77185D2F00965468A440C13F1E02D2F_used_collection	sparql_sol_working_download_to_pdf.png_f77298b
pdflatex_F77185D2F00965468A440C13F1E02D2F_created_collection	RRoth_Thesis.pdf_a2816ef
pdflatex_5691E5EFD0F6FD64835D5D293C7C8A42_created_collection	RRoth_Thesis.pdf_3da3617
pdflatex_5691E5EFD0F6FD64835D5D293C7C8A42_used_collection	RRoth_Thesis.tex_470c35c
pdflatex_F77185D2F00965468A440C13F1E02D2F_used_collection	RRoth_Thesis.tex_94cd06f
tar_2EB94E05F0FE390CA15871909D078506_created_collection	RRoth_Thesis.tex_470c35c
tar_2EB94E05F0FE390CA15871909D078506_created_collection	RRoth_Thesis.pdf_1a5e489

Figure 4.26: SPARQL query for provenance of multiple pdflatex executions



SPARQL query:

```
 PREFIX prov: <http://www.w3.org/ns/prov-o#>

SELECT DISTINCT  ?collection ?lib
WHERE {
        ?collection prov:hadMember ?lib .

        filter not exists {
                ?collection2 prov:hadMember ?lib, ?otherLib .
                filter ( ?collection2!= ?collection)
        }
        FILTER(regex(str(?collection),"lib"))
        FILTER(regex(str(?lib),"gcc|nixos|fontconfig|glibc"))
}
```

collection	lib
pdflatex_FDE3ED129D21D9A5D32FA8C66FA1F47A_lib_collection	/nix/store/dk0n769l985raba2nrya2q7ivspafj6f-gcc-6.4.0-lib/
pdflatex_FDE3ED129D21D9A5D32FA8C66FA1F47A_lib_collection	/nix/store/fw47n9vn1fz7nxdwagmipwyxmp9rxxpv-nixos-system-nixos-17.09.2905.c1d9aff56e0/
pdflatex_FDE3ED129D21D9A5D32FA8C66FA1F47A_lib_collection	/nix/store/1i4m56h236dlvmgkqr4063jll07sds1v-fontconfig-2.12.1-lib/
pdflatex_FDE3ED129D21D9A5D32FA8C66FA1F47A_lib_collection	/nix/store/d54amiggq6bw23jw6mdsgamvs6v1g3bh-glibc-2.25-123/
pdflatex_F77185D2F00965468A440C13F1E02D2F_lib_collection	/nix/store/y5ac95kk3nb52si8zcyznjrfb45720hk-gcc-6.4.0-lib/
pdflatex_F77185D2F00965468A440C13F1E02D2F_lib_collection	/nix/store/rj8nnlz5w3aa8rpcfb5fa7km29mc3lr4-fontconfig-2.12.1-lib/
pdflatex_F77185D2F00965468A440C13F1E02D2F_lib_collection	/nix/store/2ppk47fd7xnbf4pspzgp3l1zhijxgafv-nixos-system-nixos-17.09.1756.c99239bca0/
pdflatex_F77185D2F00965468A440C13F1E02D2F_lib_collection	/nix/store/xzx1bv1d7z4mgg6sg6ly0jx609qvka4x-glibc-2.25-49/

Figure 4.27: SPARQL query for provenance of multiple pdflatex executions after NixOS upgrade

CHAPTER 5

Critical Reflection

In this chapter we look to answer the questions posed in section 1.3. To do so, we will reflect on the fundamental goals that utilities enabling the reproducibility of computational research seek to accomplish. Furthermore, we will explore in what ways our presented solution system succeeds in meeting these goals, as well as the obstacles faced while doing so. From these observations, we will then suggest applications for which our proposed solution could be viably employed.

5.1 Provenance for Reproducibility

Provenance is critical for the definition of reproducibility as used within the context of this thesis. It allows one to trace the origin of data though all steps in which has been modified, exchanged, and created. In the development of our proposed solution, we looked to answer the question about what constitutes adequate provenance information required for reproducibility. The answer to these questions is best represented though our chosen model implemented using the PROV-O ontology as seen in Figure 4.6. The object properties, data properties, annotations, and literals capture the required data and present it in such a way that its structure is easily accessible to the average and skilled user alike. Importantly, these data points allow the user to identify exactly what resources were used and/or created, when these resources were accessed, and where they are located on the host file system for all process executions contained within the ontology.

As we have discovered during the course of the presented work, capturing the required information to address these questions requires special attention. Specifically, resources can have existed in a variety of states on the host system and the ability to not only reference, but to retrieve the correct variant of the resource essential to the task of reproducing work via provenance information. To do so we require unique identifying information provided from, as in our proposed solution, version tracking applications and functional programming paradigms.

The final critical component is the attachment of meaning to the provenance information. Without meaning, the data for which we have provided the above data points for lacks purpose and function. To do so, the data must be related to a user/agent who performed a specific action. This affiliation between the data and a user/agent allows for the inference of much more intrinsic relationships to be discovered. For example, justifications for the actions performed can be deduced, responsibility and the ownership of data can be designated, and accountability can be ascertained.

5.2 Captured Data and its Volume

In this section we look to reflect on what data was producible from the methodologies outlined in Chapter 3. In fact, the difficulties faced during development were not in producing the required data, but in reducing the amount of data produced to only the relevant and necessary information. These obstacles were both as a result of and remedied by the Linux Audit Daemon.

Developing the rules and configuration of auditd was both a time consuming and critical exercise in the development of the proposed solution. Close attention had to be paid to the rules in order to ensure not only that the desired information was captured, but that the amount of captured information remained feasible to extract and parse. As an example of this, consider the first rule in Figure 4.3. Notice that this rule monitors the *open* and *openat* syscalls, but only those specifying the “write” permission type. This might seem counterintuitive, but it is with good reason that we chose to monitor open syscalls as opposed to write syscalls. Simply put, monitoring the write syscalls has the consequence of flooding the auditd log file, often times with redundant information, that may result in the logs containing more information that can be timely processed. Subsequently, the log files would grow quicker than the auditd log entries could be parsed, and provenance information would be lost. Furthermore, the ordering of the rules is also important when considering the keys assigned to each. As entries are processed, they will match with the first matched rule. This can result in a vastly different categorization of data given to the Auditd Log Parser should the ordering of the rules be "incorrect", and subsequently incorrect parsing of the provenance information.

An additional design decision to curb the amount of data being logged was to initiate both auditd and the Auditd Log Parser application from an account different from that on which users will be performing their work. This allows for the specification of rules, as seen in Figure 4.3, indicating to ignore all auditd log entries not from a specific UID. In this way, we can automatically filter out all entries resulting from the Auditd Log Parser itself without having to do any processing of our own. This results in drastically smaller log files and quicker parsing times. Additionally, we observed in section 4.4.3 how failing to exclude log entries from other UIDs can result in log files growing quicker than they can be processed by the Auditd Log Parser. As a consequence of specifying the UID exclusion rules as seen in Figure 4.2, the solution system captures the provenance of only a single UID.

We must also consider the amount of information we wish to retrieve from the auditd logs. As we can see in Figure 3.11, each auditd log entry contains much more information that what has been utilized in the solution system. For example, the fields a0, a1, a2, and a3 contain detailed information about the arguments passed to corresponding syscall. These fields are encoded in hexadecimal notation, and are interpretable with utilities specially designed for auditd, such as the ausearch utility. There are numerous other fields found in the auditd log entries, and we encourage the readers to further explore them and their applications.

Finally, we must take into account the effects of collected volume of data on the host system. It should not be surprising that a high volume of entries being written to a log file requires extensive computing power. In fact, if one were to start auditd with a single rule to monitor all syscalls, the entire system could be rendered unusable. It is therefore important to that the configuration of the solution be suitable to the host system on which it is running, and that the host system have sufficient computing power.

The culmination of the above points allows for our solution system to be a feasible approach for capturing and manipulating provenance information. Without these consideration, logs files would grow to unmanageable sizes within seconds and the amount of parsing would be detrimental to the overall runtime and performance of the solution system. Thus, acquiring a sufficient amount of data to represent the provenance of the system is not an issue. What is an issue however, is regulating the flood of information and sifting through it to identify the relevant and necessary data points.

5.3 Captured Data Representation and Accessibility

In this section we address some of the obstacles faced in representing the provenance information and making it accessible to users. One of the key results of the solution system is that is is easy to use for users of all skill levels. Being successful in this regard required numerous decisions with respect to what format the information should be represented in, and what applications can and should be used for interaction with the information.

A critical component for the accessibility of the information was to allow for graphical interaction. There are numerous applications and APIs available that allow for the representation of ontologies, but it was ultimately decided to use Stanford's Protégé OntoGraf and SPARQL Query plugins. The reasons for this decision included ease-of-use and a full-suite of ready-made features. While all of the captured provenance information is accessible through the SPARQL Query plugin, this is not the case for OntoGraf. Specifically, OntoGraf was not able to graphically represent the object property annotations. For our solution, this means that the *xsd:dateTime* literal indicating the time a resource was used by a process is not observable with the OntoGraf plugin. However, using "Individuals by class" tab in the Protégé application allows for this information to be accessed. Furthermore, this information is also accessible through the SPARQL Query plugin, as seen in Figure 4.21.

5.4 Viable Applications

The beauty of the presented solution system is that it is applicable to just about any task performed on the host system. This is due to the functionality provided by the combined use of the NixOS, the Linux Audit Daemon, and the Auditd Log Parser. Data within the monitored directory is capable of being traced back to the application that used and or generated it, as well as to the responsible user. This opens a wide array of possibilities, a few of which we will cover in this section.

In section 2.1.3 we examined the Freesufer application used to analyze MRIs of the human brain. As discussed, it was often the case that the measurements being output by Freesurfer were inconsistent between systems. While this was sometimes due to hardware differences, the underlying OS and utilized version of Freesurfer played a major role in contributing to these differences. In [GHJ$^+$12] it was suggested that users provide information about the employed version of Freesurfer, as well as information about the OS and workstation. The work presented in this paper would allow for the automated collection of this data, as well as the parsing of this data into ontological form. This would provide future users with a more complete overview of the original working environment from which they could identify crucial information allowing for accurate comparison of results from computational research applications.

Our solution system can also be used to enable extensive user auditing on the host system. Specifically, though both OntoGraf and SPARQL queries, it is possible to query the ontology for all users that have interacted with a given resource. The key word in the previous sentence is "interacted", as this includes not only files that were modified by the user, but also those that were utilized by processes that the user executed. This allows for users obtain granular data time-lines regarding data access and modifications, as well as to infer a resources origins.

Additionally, the solution system can also be used to enable extensive data auditing on the host system. In much the same way that user auditing can be performed to analyze all users that have interacted with a given resource, we can examine all executables, processes, and resources that been utilized by a user or executable. This is useful in situations where perhaps one is interested in resolving dependency information, or to what extent a resource is utilized by applications on the the host system. This information lends itself well to tasks such as determining the complexity of a process and its closure with respect to its utilized resources and artifacts. In turn, the closure of a process can be used to verify the process' accuracy, integrity, and authenticity.

5.5 Future Work

After having considered our reflections and results, in this section we explore what areas are still open to future work and development.

Extensions to the Linux Auditing System

In this presented solution, our Auditd Log Parser and Ontology Maintenance components are custom developed solutions that run on top of auditd. In a future solution system such as that which we presented, it would be beneficial to integrate these components into the Linux Auditing System, and subsequently the kernel. Configuration parameters could be specified for the parser, similar to those in Figure 4.3, detailing what fields should be parsed from the log entries. Additionally, parameters indicating how to represent these fields in an ontology would be configurable. As the OWL encoding of ontologies are nothing more than text files, they could be supplied to the Linux Auditing as part of the configuration file. In effect, this would allow for the automatic creation of ontologies from the auditd log files, while only requiring lightweight configuration from the user.

Standalone Graphical Interface

While investigating the available options allowing for the graphical interaction with ontologies, it became clear that a viable solution is missing from the academic market. There are plenty of available API's allowing for the importing of ontologies and subsequent graphical representation, but lacking are ready-made solutions. Stanford's Protégé also offers an API allowing developers to implement plugins as standalone applications, but this process is not straight forward and documentation is out of date. A graphical representation of the ontology seems to be the most approachable method for users of all skill levels to interact with the information. As such, the the development of a standalone application allowing for such functionality would bring ontologies to a much larger audience.

Workflow Integration

Workflows, as discussed in section 2.2.1, undoubtedly offer increased reproducibility of computational research results. We make two observations: (1) WFMSs do not typically provide the means to inspect the provenance of individual data required for execution, and (2) the presented solution does not present the procedure that was employed during the execution of some (multi-step) task. Combining these two points into a single solution would allow for a more detailed representation of the actions that were performed on a system, while providing the users with granular provenance information for the utilized resources.

5.6 Summary

In this chapter we reflected on some of the overarching themes and concepts of the presented work, as well as a handful of obstacles encountered during development.

The components making up the provenance information required for reproducibility were covered in section 5.1, where we emphasized the importance of establishing the relationships between the provenance data points. Furthermore, in section 5.2 we explored many obstacles related to the pure volume of data being captured. These obstacles ranged from possessing enough computing power and storage, to feasible auditd configurations. Additionally, we reviewed some of the shortcomings and advantages of Stanford's Protége plugins OntoGraf and SPARQL Query in section 5.3, and the viable applications of the solution system in section 5.4. Finally, in section 5.5, we outlined various areas in which the presented work can be expanded.

CHAPTER 6

Conclusion

In this thesis, we explored the many challenges that researchers face when attempting to reproduce computational research results, and how our proposed solution addresses these challenges. These challenges range from inconsistent results due to various hardware and software environments, to missing input data critical to the execution of some task. Our solution addresses these issues by automatically capturing and parsing system-wide provenance information into ontological form. In this form, users are able to interact with the ontology through applications such as Stanford's Protégé. In our working example, we demonstrated how the Protégé plugins OntoGraf and SPARQL Query can be used to access very granular provenance information for a given file, including the provenance of its contents such as images and tables. We then exhibited how the ontological form of this information can be used to access information about the pdflatex executable from which this file was created, such as additional generated files, as well as utilized resources and libraries. Additionally, detailed information such as file access and generation times, file locations and versions, and how the relationships between the ontology's entities, activities, and agents are made easily accessible to the user through graphical representation. It is through this ontological representation of the provenance information that researchers gain a better and more accurate description of the working environment and the data both created and utilized by a given task or entity. This in turn allows researchers to more reliably and effectively reproduce previous computational research results both of their own and of their peers.

List of Figures

List of Tables

Acronyms

API Application Programming Interface. 44, 47, 69, 71

AUID Audit User ID. 43

CWD current working directory. 32

DBS Database Management System. 11

DLL Dynamic Linked Library. 22

GUI Graphical User Interface. 34, 58–60

IRI Internationalized Resource Identifier. 21, 22, 39

IT Information Technology. 37

LXC Linux Container. 13

MD5 Message Digest 5. 48, 49

MRI Magnetic Resonance Imaging. 9, 70

OS Operating System. 2, 9, 10, 13, 14, 17, 27, 38, 70

PDF Personal Document Format. 2, 34, 48, 49, 54–58

PID Process ID. 48

RDF Resource Description Framework. 17, 19, 20, 32, 33, 35

SHA-1 Secure Hash Algorithm 1. 29, 56

SVN Subversion. 28

UID User ID. 32, 40, 43, 47, 58, 60, 68

URI	Uniform Resource Identifier. 18–21
URL	Uniform Resource Locator. 55
VCS	Version Control Software. 28, 35
VM	Virtual Machine. 5, 13, 14, 16, 38
WFMS	Workflow Management System. 11, 71
WWW	World Wide Web. 18, 35
XML	Extensible Markup Language. 39, 55

Bibliography

[Apa17] Apache. Apache Jena: A free and open source Java framework for building Semantic Web and Linked Data applications, 2017.

[BDW16] John D. Blischak, Emily R. Davenport, and Greg Wilson. A Quick Introduction to Version Control with Git and GitHub. *PLOS Computational Biology*, 12(1):e1004668, January 2016.

[BLHL01] Tim Berners-Lee, James Hendler, and Olli Lassila. The Semantic Web" in Scientific American. *Scientific American Magazine*, 284, 2001.

[BM04] Paul V. Biron and Ashok Malhotra. XML Schema Part 2: Datatypes Second Edition. W3C Recommendation, W3C, October 2004.

[Boe15] Carl Boettiger. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79, January 2015.

[Bra07] Steve Bratt. Semantic Web, and Other Technologies to Watch. Presentation at the 2007 INCOSE International Workshop, Albuquerque, New Mexico, January 2007.

[CC14] Bo Chen and Reza Curtmola. Auditable Version Control Systems. 2014.

[CPM$^+$14] Christian Collberg, Todd Proebsting, Gina Moraila, Zuoming Shi, and Alex M Warren. Measuring Reproducibility in Computer Systems Research. In *1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*, 2014.

[CWL14] Richard Cyganiak, David Wood, and Markus Lanthaler. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation, W3C, February 2014.

[DdJV04] Eelco Dolstra, Merijn de Jonge, and Eelco Visser. Nix: A Safe and Policy-Free System for Software Deployment. *Proceedings of the 18th USENIX Conference on System Administration*, pages 79–92, 2004.

[DLP10] Eelco Dolstra, Andres LöH, and Nicolas Pierron. NixOS: A purely functional Linux distribution. *Journal of Functional Programming*, 20(5-6):577–615, November 2010.

[Dru09] Chris Drummond. Replicability Is Not Reproducibility: Nor Is It Good Science. *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*, 2009.

[Fal16] Sean Falconer. OntoGraf, 2016.

[GHJ+12] Ed H. B. M. Gronenschild, Petra Habets, Heidi I. L. Jacobs, Ron Mengelers, Nico Rozendaal, Jim van Os, and Machteld Marcelis. The Effects of FreeSurfer Version, Workstation Type, and Macintosh Operating System Version on Anatomical Volume and Cortical Thickness Measurements. *PLoS ONE*, 7(6):e38234, June 2012.

[GRa] Steve Grubb and Redhat. auditctl(8) - Linux man page.

[GRb] Steve Grubb and Redhat. auditd(8) - Linux man page.

[GRc] Steve Grubb and Redhat. auditd.conf(5) - Linux man page.

[GRd] Steve Grubb and Redhat. audit.rules(7) - Linux man page.

[GSP13] Steve Harris Garlik, Andy Seaborne, and Eric Prud'hommeaux. *SPARQL 1.1 Query Language.* W3C Recommendation. W3C, 2013.

[HB09] Matthew Horridge and Sean Bechhofer. The OWL API: a java API for working with OWL 2 ontologies. In *Semantic Web*, volume 2, January 2009.

[HBC15] Daniel G. Hurley, David M. Budden, and Edmund J. Crampin. Virtual Reference Environments: a simple way to make research reproducible. *Briefings in Bioinformatics*, 16(5):901–903, 2015.

[HTT09] Tony Hey, Stewart Tansley, and Kristin Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery.* Microsoft Research, October 2009.

[HWS+06] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R. Pocock, Peter Li, and Tom Oinn. Taverna: a tool for building and running workflows of services. In *Nuclear Instruments and Methods in Physics Research A*, 2006.

[ISO04] ISO8601. Data elements and interchange formats – Information interchange – Representation of dates and times. Standard, International Organization for Standardization, Geneva, CH, December 2004.

[Kuc04] G Kuck. Tim Berners-Lee's Semantic Web. *South African Journal of Information Management*, 6, 2004.

[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: Research Articles. *Concurrency and Computation: Practice & Experience*, 18(10):1039–1065, August 2006.

[LMS13] Timothy Lebo, Deborah McGuinness, and Satya Sahoo. PROV-O: The PROV Ontology. W3c Recommendation, W3C, April 2013.

[MM13] Paolo Missier and Luc Moreau. PROV-DM: The PROV Data Model. W3c Recommendation, W3C, April 2013.

[MP12] Boris Motik and Bijan Parsia. OWL 2 Web Ontology Language Document Overview (Second Edition). W3C Recommendation, W3C, December 2012.

[MPS+13] Tomasz Miksa, Stefan Pröll, Stephan Strodl, Ricardo Vieira, José Barateiro, and Andreas Rauber. Framework for Verification of Preserved and Redeployed Processes. *iPRES 2013 - 10th International Conference on Preservation of Digital Objects*, pages 136 – 145, 2013.

[MR15] Rudolf Mayer and Andreas Rauber. A Quantitative Study on the Reexecutability of Publicly Shared Scientific Workflows. In *11th International Conference on e-Science*, pages 312–321. IEEE, August 2015.

[MRM16] Tomasz Miksa, Andreas Rauber, and Eleni Mina. Identifying Impact of Software Dependencies on Replicability of Biomedical Workflows. *Journal of Biomedical Informatics*, 64, 2016.

[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 *Scientific and Statistical Database Management: 22nd International Conference, SSDBM 2010, Heidelberg, Germany, June 30–July 2, 2010. Proceedings*, pages 471–481. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. DOI: 10.1007/978-3-642-13818-8_33.

[Mus15] Mark A. Musen. The protégé project: a look back and a look forward. *AI Matters*, 1(4):4–12, June 2015.

[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, 2004.

[Pen11] R. D. Peng. Reproducible Research in Computational Science. *Science*, 334(6060):1226–1227, December 2011.

[PF16] Stephen R Piccolo and Michael B Frampton. Tools and techniques for computational reproducibility. *GigaScience*, 5(1):1–13, 2016.

[PSPM12] Peter Patel-Schneider, Bijan Parsia, and Boris Motik. OWL 2 Web Ontology Language Structural Specification and Functional-Style Syntax (Second Edition). W3C Recommendation, W3C, December 2012.

[Rad12] Tijs Rademakers. *Activiti in Action : Executable business processes in BPMN 2.0.* Manning Publications, Shelter Island, NY, first edition, 2012.

[Red] Redhat. Chapter 6. System Auditing.

[Red16] Timothy Redmond. SPARQL Query, 2016.

[Rom08] Paolo Romano. Automation of in-silico data analysis processes through workflow management systems. *Briefings in Bioinformatics*, 9(1):57–68, 2008.

[SBO$^+$07] Sören Sonnenburg, Mikio L. Braun, Cheng Soon Ong, Samy Bengio, Leon Bottou, Geoffrey Holmes, Yann LeCun, Klaus-Robert Müller, Fernando Periera, Carl Edward Rasmussen, Gunnar Rätsch, Bernhard Schölkopf, Alexander Smola, Pascal Vincent, Jason Westen, and Robert Williamson. The need for open source software in machine learning. *Journal of Machine Learning Research*, 8, October 2007.

[SNTH13] Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. Ten Simple Rules for Reproducible Computational Research. *PLoS Computational Biology*, 9(10):e1003285, October 2013.

[Spi12] D. Spinellis. Package Management Systems. *IEEE Software*, 29(2):84–86, March 2012.

[The17] The Eclipse Foundation. JGit, 2017.

[Tor17] Linus Torvalds. Git, 2017.

[WFDRG09] Katy Wolstencroft, Paul Fisher, David De Roure, and Carole Goble. *Scientific Workflows.* OpenStax CNX, November 2009.