A FRAMEWORK FOR REPRODUCIBLE COMPUTATIONAL RESEARCH

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

BY
QUAN TRAN PHAM

CHICAGO, ILLINOIS
AUGUST 2014
# TABLE OF CONTENTS

LIST OF FIGURES .......................................................... v
LIST OF TABLES ............................................................ vii
ACKNOWLEDGMENTS ......................................................... viii
ABSTRACT ................................................................. x
TERMINOLOGY .............................................................. xi

1 INTRODUCTION .......................................................... 1
  1.1 Growth of Reproducible Scientific Research ......................... 2
  1.2 Challenges of Making Computation-based Research Reproducible . . . 4
  1.3 Problem Overview .................................................... 5
    1.3.1 (G1) Develop a methodology and framework to support authors and
          readers in the conduct of reproducible research ................. 6
    1.3.2 (G2) Increase the efficiency of reproducible research process for
          authors, while making it easy for readers to conduct verification at
          various granularity levels ........................................ 7
    1.3.3 (G3) Make the framework generalizable to many computational envi-
          ronments .......................................................... 7
  1.4 Solution Overview .................................................... 7
  1.5 Solution Demonstration ............................................... 10
  1.6 Dissertation Roadmap ............................................... 16

2 RELATED WORK .......................................................... 18
  2.1 Computational Artifact Publication and Usage ......................... 18
    2.1.1 Inside Publication; Static Verification .......................... 19
    2.1.2 Inside Publication; Dynamic Verification ......................... 19
    2.1.3 Outside Publication; Static Verification ......................... 20
    2.1.4 Outside Publication; Dynamic Verification ......................... 22
  2.2 Computation Artifact Creation ........................................ 24
    2.2.1 Software Approaches ............................................. 24
    2.2.2 Virtual Machine Approach ....................................... 25

3 SOLE FRAMEWORK ........................................................ 27
  3.1 Publication and Purpose ............................................... 27
  3.2 Science Objects ..................................................... 28
  3.3 The SOLE Environment ............................................... 31
  3.4 Tools in the SOLE Reproducible Research Framework ................. 32
    3.4.1 Language Object Creation Tool .................................. 34
    3.4.2 Data Object Creation Tools ..................................... 36
# LIST OF FIGURES

1.1 SOLE framework architecture ..................................................... 8
1.2 Components of a computational research paper ............................... 11
1.3 CTagsCrawler generates a language science object from R source code. Source code is annotated by authors with tags (left) that will be associated with text from the paper (right) ......................................................... 11
1.4 Science objects get associated to text in a research paper ................. 12
1.5 PDFExtractor creates a data science object from referenced data in another paper. The extracted text from the referenced paper (top). The highlighted value $1 per MWh refers to the annotation science object with the tag Texas (bottom) 13
1.6 A browser history science object shows paths and keywords used to search for data set NCI ................................................................. 13
1.7 PEEL\(_0\): a three-step workflow experiment ................................ 14
1.8 Using provenance to capture and re-execute in the PTU tool ............... 15
1.9 Explore and re-execute a repeatable software package using an GUI interface in the SOLE framework ............................................. 16

2.1 Related work classification using Efficiency (minimal incentives), Usability (multiple reader purposes), and Complexity (complex computation environments) to address challenges in Section 1.2 ................................. 23
2.2 CDE audit and execution modes .................................................. 24

3.1 The SOLE Architecture ............................................................. 32
3.2 The steps that an author follows to create and associate science objects in SOLE ................................................................. 33
3.3 Enhancing conceptual description with Language Science Objects ........ 37
3.4 Annotating a PDF by selecting a rectangular region and adding tag Texas 39
3.5 The extracted text from the paper (top). The highlighted value $1 per MWh refers to the annotation science object with the tag Texas (bottom) ........... 40
3.6 A browser history science object shows paths and keywords used to search for data set NCI ................................................................. 42
3.7 First 10 links of the path from the beginning of the browser session to the downloaded dataset step in the end ........................................ 43
3.8 Distinct link used in the browser session ....................................... 44
3.9 Search terms used in the browser session ....................................... 45

4.1 Author: ptu-audit uses CDE to build a package of provenance and file dependencies ................................. 51
4.2 Tester: views the provenance graph stored in the database of the PTU package ................................................................. 52
4.3 ptu-exec restarts from /bin/calculate without re-executing /bin/workflow, uses CDE to re-route file dependencies ........................................ 52
4.4 Example of using PTU configuration file to control file inclusion ........ 57
4.5 PTU configuration file to control different inputs ............................. 59
4.6 Provenance graph of PEEL\(_0\) program ........................................ 59
4.7 Time reduction in testing PEEL\(_0\) using PTU ................................ 60
4.8 Time reduction in testing TextAnalyzer using PTU ........................... 61
4.9 Software Packages of A, B, and C .................................................. 67
4.10 A’s package is reused in C’s package. PTU generates a subgraph “f4 → Aggregation →
f5 → Generate Image → f6” from C’s package. Since that subgraph is sub-
graph isomorphic to the provenance graph in A’s package, PTU can validate A’s
authorship on a part of C’s package. .................................................... 70
4.11 Overhead when using CDE with Kameleon VM appliance ...................... 73
4.12 SOLE PDF Reader interface for provenance exploration: an experiment run two
processes “process.py” in two remote machines, used scp to copy their outputs
back to local machine, then execute “summary.py” to generate the final output. 79
4.13 SOLE PDF Reader interface for exploring PDF-format paper with attached
provenance: clicking on some text in the paper can trigger the Provenance Ex-
plorer interface to show a corresponding artifact. .................................... 81
4.14 Re-execute a process in SOLE PDF Reader ..................................... 83

5.1 Socket node stores a 5-tuple and socket transferred data ......................... 89
5.2 SSH injection ................................................................................. 93
5.3 Merge local provenance databases .................................................. 95
5.4 Provenance graph with Socket node and transparent SSH process dependency . 98
LIST OF TABLES

1.1 Reader purposes for studying a scientific publication ........................................ 5

2.1 Classification of scientific publication computational artifacts ............................ 18

4.1 LevelDB key-value pairs that store file and process provenance. Capital letter words are arguments. ............................................................... 65

4.2 Ratio of different files having the same path in 5 popular AMIs. The denominator is number of files having the same path in two distributions, and the numerator is the number of files with the same path but different md5 checksum. Ommited are manual pages in /usr/share/ directory. ........................................ 68

4.3 Increase in PTU-SP performance is negligible in comparison with CDE ............ 72

4.4 Provenance edge in SOLE PDF Reader’s Provenance Explorer interface ............. 80

5.1 Network system call audit and replay: bold parameters and bold returned values are recorded during audit and injected back for replay ........................................ 91

5.2 NASA Parallel Benchmark runtime (seconds) and overhead of PTU meta audited mode (for query) and content audited mode (for replay) compared to normal NPB execution (no PTU) .................................................. 97

5.3 PTU performance (seconds) on network related and non-network related tasks from a multi step experiment shows 3-fold reduction in re-execution time with network replay. ......................................................... 99
ACKNOWLEDGMENTS

This dissertation could not have been possible without the support and encouragement of many marvelous people.

First and foremost, I would like to thank my advisor Prof. Ian Foster for the continuous support of my Ph.D. study and research, for his patient guidance and immense knowledge. He has been a great source of information to help me through the whole process of research and writing of this thesis.

I would like to thank Dr. Tanu Malik, my thesis co-advisor. She has been a great inspiration that guide me through all the steps to finish my Ph.D. study. We started our collaboration in 2011 since when ideas for this thesis began to materialize. I am really grateful with her guidance in research process, developing ideas and writing papers. Her insights on my thesis have helped to create the system that we have today.

I would also like to thank my dissertation committee member Prof. Anne Rogers for her encouragement, insightful comments, and suggestions.

I owe sincere thankfulness to Neil Best, he is the author of the first research paper to which our system applied. He has spent a lot of time working with me towards the foundation of ideas in my thesis. I would like to thank the following participants in the RDCEP Center, in particular, Joshua Elliott and Justin Wozniak at The University of Chicago, Columbia University, and Argonne National Laboratory for motivating my use cases, Allison Brizius for describing the Center’s activities, Jonathan Ozik for sharing their paper and code.

I thank my fellow lab mates. They are the best graduate students around. Atilla Soner Balkir, Zhao Zhang, Gabriella Turcu, Jing Tie, and Allan Espinosa, you have made my Ph.D. life a wonderful experience. I want to thank Ioan Raicu for the time he guided the new student to the academic environment.

Last but not least, I would like to thank my family. First, my parents and my brother for their phenomenal support and their belief they have in me. I would like to thank my
wife, Van Thi Quynh Nguyen, for her love, her patience, and her support that makes this happens. And to my daughter, Lexi, who makes everyday a wonderful day.

This work is supported by the University of Chicago’s Department of Computer Science, the Computation Institute of the University of Chicago and Argonne National Laboratory, and the Center for Robust Decision making on Climate and Energy Policy under NSF grant number SES-0951576 and subcontract award under grant GEO-1343816.
ABSTRACT

In today’s world of publishing, reproducing research results has become challenging as scientific research has become inherently computational. Encoding a computation-based result in a text-based paper is nearly impractical, leading to the overarching research question. “Can computation-based research papers be reproducible?”

The aim of this thesis is to describe frameworks and tools, which if provided to authors can aid in assessing computational reproducibility. Towards this aim, the thesis proposes a reproducible framework Science Object Linking and Embedding (SOLE) for creating descriptive and interactive publications by linking them with associated science objects, such as source codes, datasets, annotations, workflows, process and data provenance, and re-executable software packages. To create science objects in a linkable representation for use within research papers, the thesis describes a set of tools as part of the framework. In particular, it focuses on Provenance-To-Use (PTU), an application virtualization tool that encapsulates source code, data, and all associated data and software dependencies into a package. We describe how by capturing data dependencies, PTU allows full and partial repeatability of the virtualized software; and by capturing software dependencies, PTU can be used for building and maintaining software pipelines. Finally, we show how PTU can be used to provide computational reproducibility in a distributed environment.

We evaluate and validate the framework by applying it to several representative publications and determining the extent to which computational reproducibility is achievable.
TERMINOLOGY

• Code - software source code, binary, software library, and/or script that consists of defined instructions to instruct hardware to perform the tasks for which it is designed.

• Data - input file, input parameter that is needed for Code to produce output of a computational experiment; intermediate file and final output are also Data.

• Environment (in a context of capture and replay) - environment variables, part of the operating environment in which a process runs, can affect the way running processes will behave on a computer.

• Execution - a process by which a computer or a virtual machine performs the instructions from some Code, that uses Data and Environment specified by an experiment.

• Verification - The evaluation of whether or not a software, service, or system complies with a regulation, requirement, specification, or imposed condition.

• Validation - The assurance that a software, service, or system meets the needs of identified stakeholders.

• Scientific Reproducibility - verification and validation of a scientific claim by other scientists (stakeholders).

• Computational Reproducibility - Given all the computational artifacts associated with a scientific experiment, the possibility of (i) repeating the scientific experiment with same inputs and obtaining exactly same outputs, or (ii) repeating with configuration/parameter changes and obtaining predictable and similar output, and/or (iii) extending the experiment.

• Reproducible Paper - an article in which all necessary and sufficient information has been provided to verify and validate claims and results.
CHAPTER 1
INTRODUCTION

There are two goals for scientific publications: to announce a result and to provide sufficient evidence that the result is correct. Numerous examples, however, demonstrate that provided evidence to verify and validate results is not sufficient. For example, a cancer researcher, C. Glenn Begley, studied 53 “landmark” publications papers in top journals from reputable labs and found that 47 could not be repeated [6]. In a comprehensive study over medical research findings, Ioannidis demonstrated through simulation that most research studies are based on a single study, conducted in specialized settings with narrowly scoped study designs and therefore it is more likely for claims to be false than true [40] in any real setting. Consequently, several research communities have called for scientific experiments in papers to be described and adhered to at least a replication standard, advocating the sharing of research information beyond that included in the traditional publication: “The replication standard holds that sufficient information exists with which to understand, evaluate, and build upon a prior work if a third party can replicate the results without any additional information from the author” [43, 57].

Establishing specification for such a replication standard has become challenging as research has become inherently computational. Prior to the computation-driven revolution in science, the replication standard was easily met in research papers with the papers describing experiments involving small amounts of data, derivations on that data, and associated methods and algorithms. Readers reproduced the results by repeating the physical experiment, performing hand calculations, and/or logical argument.

The scientific method in this decade has become decisively computational, frequently involving large quantities of data, complex data manipulation tasks, and large, and often distributed, software stacks. Encoding a computation-based replication standard in a text-based paper is nearly impractical. Therefore current research papers summarize the
associated data and computation rather than reproduce it computationally. To provide valid evidence, research papers currently adopt indirect means, such as building companion websites that share data and software packages. However, these external websites remain disconnected from the content within the paper, making it difficult to verify claims and reproduce results. This disconnect oftentimes affects readability of a computation-based publication and limits readers and reviewers from being able to assess the validity of findings, determine workability of results on different parameters, or reuse data and methods for their own research. In fact for most researchers, it is an implied understanding in scientific domains that a true assessment of a computation-based publication is only feasible if readers have in-depth knowledge of the computation-based tools used in the research and an out-of-band communication channel with the authors.

This leads to the larger question: “Can research papers describing a computational scientific experiment be made reproducible?” Towards this question, we propose the following overarching thesis: “Research papers describing computational scientific experiments can be made reproducible if authors and readers are provided with tools to capture, verify and validate the experiment”. This thesis claim assumes that a computation-based research paper primarily consists of scientific experiments that can be described in terms of computational methods and computational artifacts. A computational method is equivalent to an algorithm, recipe or procedure, and a computational artifact may imply source codes, data, binary programs, run-time environments, workflows, and operating system and hardware specifications.

1.1 Growth of Reproducible Scientific Research

The sharing policies of 170 journals revealed 3.5% journals required code sharing as a condition of publication, and 11.2% journals required data sharing as a condition of publication [91]. This survey was carried out on two years, 2011 and 2012. The authors concluded that
code and data sharing policies were not widespread, but are being adopted rapidly with the biggest change makers being journals with high impact factors.

To support these changes, a vast number of reproducible tools have been introduced to different science domains. To name a few: Cloud BioLinux [47] provides pre-configured and on-demand computing for the genomics community, the Galaxy [32] web-based platform is popular in biomedical domain; GenePattern [82] is a reproducible bioinformatics platform; Research Compendia [85] is for economists; climate scientists use VisTrails [5] for simulations, data exploration and visualization; and finally CDE [35, 36], Collage Authoring Environment [65], MyExperiment [31], Sumatra [21], Sweave [52], and Taverna [66] have claimed to solve reproducibility across science domains.

We review these tools and their strengths and limitations in Chapter 2. The results of this survey provided us with two key insights: (i) several tools provide verification and not validation, and (ii) most tools are limited to local computational environments and cannot be extended to distributed and remote environments, thus limiting which research papers can be made reproducible. We further observe that computational reproducibility is feasible if the primary agents, namely the authors and readers, are provided with appropriate tools that make reproducibility an effect of the research process rather than a separate process to be undertaken. In the past, this evidence was provided through literate programming approaches, such as CWeb, Sweave and more recently through GenePattern, a computational genomics environment [82] embedded into Microsoft Word [17]. However, these approaches do not generalize to all kinds of computational artifacts and methods.

These insights persuade us to design a solution that can be used for both verification and validation, and seek novel methods for verification and validation in distributed environments. If such a solution is made available, then more computational research papers can be made reproducible than otherwise ¹.

¹. An equally pertinent question is “Which papers to reproduce?” We assume that there is an implied understanding amongst a community as to which papers to reproduce.
1.2 Challenges of Making Computation-based Research Reproducible

We elucidate challenges in developing a solution for making research papers reproducible. Different communities have emphasized on one or more aspects of the challenge. For instance, librarians have emphasized the human challenge of reproducibility and proposed solutions take a best-effort approach by capturing, describing, preserving, and curating experiments according to some defined standards; the provenance community has emphasized automated provenance-tracking tools for reproducibility, and the publishing community has emphasized policy issues such as open-access, transparency and availability of code and data [30]. However, there is a general agreement amongst all communities about the following challenges associated with making a scientific publication reproducible:

1. **Minimal incentives**: Verification and validation can be time consuming. Currently there are minimal incentives to conduct reproducible research. Often, readers and authors are unwilling to change their current computational, programming or authoring environment or use a system that imposes significant overhead on the computing environment.

2. **Multiple reader purposes**: The reader can have multiple purposes towards validating the claims described in the paper, which may vary based on their need and role (See Table 1.1). For instance, some readers may want to repeat individual experiments for reviewing purposes to verify the author’s claim that the experiments are correctly implemented. Alternatively, if an author claims development of a software package, some readers may want to reuse that package in their own research.

3. **Complex computation environments**: The computation environment used in a research project may be local or distributed. A local environment may pose simpler computation challenge in terms of auditing and linking than a distributed environment.
Table 1.1: Reader purposes for studying a scientific publication

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reference</td>
<td>Be able to reference data, methods, and processes at various granularities.</td>
</tr>
<tr>
<td>Re-describe</td>
<td>Annotate the publication text with further descriptions.</td>
</tr>
<tr>
<td>Repeat</td>
<td>Execute the processes followed in the original publication in the same execution order and under the same environment.</td>
</tr>
<tr>
<td>Rework</td>
<td>Execute the processes followed in the original publication in the same execution order and under the same environment but with different parameters.</td>
</tr>
<tr>
<td>Reuse</td>
<td>Reuse and share data, method and processes or any constituent part of it.</td>
</tr>
<tr>
<td>Reproduce</td>
<td>Repeat but with different data, method, hardware, environment, etc.</td>
</tr>
<tr>
<td>Review</td>
<td>Be able to audit and validate results.</td>
</tr>
</tbody>
</table>

In addition, artifacts from a computation environment may be static or dynamic. Thus datasets and source codes may be verified by statically linking them but experiment workability can only be validated by linking a virtual machine with appropriate software packages.

These challenges mandate us to seek solutions that are generalizable and put the fewest requirements and constraints on authors and readers. We next provide a formal overview of the problem statement.

### 1.3 Problem Overview

We “determine an appropriate methodology for conducting reproducible research - one that can help authors to easily create computational artifacts, associate these artifacts with research papers and present them to readers; and allow readers to validate, reproduce and extend these artifacts from their original publication.”

To validate the thesis that such a methodology exists, the following three steps (stated as goals) are used:
1.3.1 (G1) Develop a methodology and framework to support authors and readers in the conduct of reproducible research

We propose a general-purpose, application-independent reproducible research framework for computational research studies that process and combine large and diverse longitudinal, complex, and distributed data and use computing to generate results. We assume that such studies are primarily disseminated through conference and journal publications that typically have a page limit, thereby limiting descriptions in a publication, and requiring further information from authors to repeat the published claims. For such studies, our primary objective in this goal is to provide tools that will enable (1) authors to easily create computational artifacts, associate these artifacts with research papers, present them to readers, and (2) allow readers to validate, reproduce and extend these artifacts from their original publication.

While many tools have been proposed as solutions to the problem of reproducible research in the sense of Goal G1 [5, 21, 32, 51, 66, 67, 84, 95], each such tool imposes limitations on authors such as requirement of adopting a reproducible framework prior to authoring papers, learning a new programming language or using a new authoring environment. In general, we observe that research is almost trivially reproducible if the author can be made to encode all steps in a suitable high-level programming language. But such a constraint ignores the messy, heterogeneous, and often circuitous nature of most research. We propose a reproducible system that is available to be adopted by the authors at any stage of the scientific life cycle, from hypothesis to publication. The authors must be free to use any programming languages they are comfortable with while continuing to use the framework to produce reproducible work on an ongoing basis.
1.3.2 (G2) Increase the efficiency of reproducible research process for authors, while making it easy for readers to conduct verification at various granularity levels

An author wants to publish the most information using the least effort. In contrast, a reader may want to assess claims within a paper at the granularity of research concepts and claims. Thus if the research paper describes a set of experiments, the reader must be able to verify claimed inputs and outputs and test the workability for each experiment. Alternatively if the intent is to verify feasibility of an algorithm or workflow, such conclusions can only be made if a trustworthy provenance trace of the experiments is associated with the publication. Ideally, the reader should be able to replay that algorithm or workflow using the provenance trace.

1.3.3 (G3) Make the framework generalizable to many computational environments

This goal implies generalizability for both authors and readers. Authors should be able to use the framework for traditional single-machine computational experiments as well as the distributed computational experiments that are becoming popular. Authors should be able to use the framework with traditional scripting languages, or with workflow process definition languages that distribute experiments to multiple computers. The framework should allow readers to reproduce computational artifacts from a different computing environment than that used by the authors.

1.4 Solution Overview

This dissertation presents Science Object Linking and Embedding (SOLE) [71], a Reproducible Research Framework (Figure 1.1). SOLE contains supporting tools that addresses
the above three goals.

(G1) Develop a methodology and framework to support authors and readers in the conduct of reproducible research  SOLE provides tools to conveniently link research papers with associated computational artifacts, such as source codes, datasets, and repeatable packages. SOLE provides tools that turn data, methods, and processes used in a scientific method into granular units of knowledge – termed science objects. In the SOLE framework, we have created a new flexible annotation tagging syntax to create language and data science objects. Our approach uses a minimal syntax vocabulary that differentiates itself from other approaches with predefined ontology classes [88, 94]. Our annotation syntax solution has been used to implement the following tools: CTagsCrawler for creating language objects from tagged source code, PDFExtractor for creating data objects from tagged PDF files, and RepAudit for creating browser-history data objects from web browsing sessions.
We have evaluated these tools by applying it to several representative publications and determining the extent to which reproducibility is achievable.

In addition to language and data science object support, the SOLE framework provides the Provenance-To-Use (PTU) [68, 72, 73] tool to capture provenance from a runtime execution for creating repeatable software package science objects. To our knowledge, this is the first study that uses Unix “ptrace” system call interposition to capture and replay provenance for repeatable software packages. We showed that PTU could be used for repeatability testing, creating and maintaining software pipelines, and reproducing results from research papers.

(G2) Increase the efficiency of reproducible research process for authors, while making it easy for readers to conduct verification at various granularity levels

SOLE helps to minimize author effort in creating science objects. Authors identify science objects with human-readable tags; the tools convert each tagged science object into an associated linked data object identified by a URI. For ease of management, the tags, URIs, and accompanying representation are maintained in a metadata repository: what is, in effect, a science object bibliography. To aid authors with the linking process, SOLE also provides a web interface that allows authors to associate groups of words in a research paper with one or more science object tags. Clicking a link in the text results in the display of an appropriate representation of the science object. SOLE persists these objects, which can be accessed and executed on-demand by users to validate or repeat an experiment in a publication.

As part of the SOLE reproducible research framework, PTU is a packaging tool for repeatability testing at various granularity levels. Using PTU, authors can accomplish two tasks: (1) build a package of their source code, data, and environment variables, and (2) store process and file level details about a reference execution of their system in an accompanying database. A PTU package relieves readers/testers from software deployment issues allowing for convenient distribution. Preserving a reference execution path and run-time details within
a package eases distribution of this vital information that can guide readers during the reproducing/testing phase. In particular, using a provided PTU package, readers can (1) explore the provenance graph and accompanying run-time details of the reference execution, and (2) specify the part of the provenance graph that they want to re-execute and execute it with the same or different inputs and settings.

SOLE also provides a separate application with a GUI interface for authors to link computational artifacts recorded in PTU to their Latex documents. Once the Latex documents are compiled into PDF format and published online, readers can use the SOLE PDF Reader to explore related provenance graphs and re-execute processes from those graphs.

(G3) Make the framework generalizable to many computational environments
By supporting network distributed experiments, SOLE can be applied to a wide range of computational environments. The PTU tool in SOLE is independent of any workflow system, works at the user level of UNIX operating systems, and thus can be adopted by most researchers. Other command-line tools from SOLE are easy to use and can be adopted by authors anytime – either while investigating the scientific method or post hoc after the publication is written. The authors need not change their authoring environment, but simply include packages or use SOLE tools to associate their publications to computational artifacts, similar to a bibliographic tool for computational products.

1.5 Solution Demonstration
Since reproducibility is a broad research topic, we use a research paper to show the scope of this dissertation and to demonstrate the SOLE framework.

In a paper named “Synthesis of a Complete Land Use/Land Cover Dataset for the Conterminous United States” [9], Best et al. describes the synthesis methodology to generate a new land cover dataset for the conterminous USA called PEEL0. The methodology is imple-
In this paper, we have identified several computational components (Figure 1.2). In particular, Best summarizes his PEEL$_0$ algorithm and references some source code, which implements the algorithm. He also uses a workflow, which combines several shell scripts and R scripts, to produce the final result. His paper uses several data values from other papers, which are referenced in the bibliography. We show how the SOLE framework can facilitate authors and readers on reproducing the result of this paper.
Using the newly provided annotation tagging mechanism from SOLE framework, we have created language and data science objects for the paper. First, following an annotation tagging mechanism described in Section 3.4.1, the authors put tags into the R source code of the experiment to encapsulate a portion of the source code. Next, the authors run CTagsCrawler, which is one of the SOLE tools, to generate language science objects (Figure 1.3). Finally, using a query-able interface to recall tags, the authors can easily associate these language science objects to phrases in their paper (Figure 1.4) and share the paper with readers. Readers can view the implementation of the mentioned algorithms and functions in the paper and understand its merit.

The SOLE framework provides a similar easy 3-step process to create and share referenced data as annotation science objects (Figure 1.5) and authors’ interaction as browser history science objects (Figure 1.6). These science objects, along with a layer of tools and interfaces that allow authors to share their computational artifacts and readers to verify them through interfaces, can minimize the disconnect between authors and readers.
RPS was initially met by existing hydropower rather than new construction [6]. Texas (1990) possesses such anomalously strong wind resources that development of windpower in the state could be driven largely by the federal Production Tax Credit and Investment Tax Credits with the state RPS playing a much less significant role. (The Production Tax Credit and Investment Tax Credits, henceforth “PTC”, reimburse qualifying renewable energy projects for up to 50% of the installed cost. See Appendix A.1 for further discussion.) Current Texas REC prices remain so low (~$1 per MWh, [13]) that the state RPS is not a significant subsidy for windpower in Texas, and current construction implies that Texas wind capacity will reach its 10 GW target almost fifteen years ahead of RPS-mandated requirements [14]. Prediction of the expected evolution of renewables implementation

Figure 1.5: PDFExtractor creates a data science object from referenced data in another paper. The extracted text from the referenced paper (top). The highlighted value $1 per MWh refers to the annotation science object with the tag Texas (bottom)

Figure 1.6: A browser history science object shows paths and keywords used to search for data set NCI
The SOLE framework also facilitates reproducibility by supporting interactive science objects. We show how the PEEL\textsubscript{0} experiment can be captured to create a repeatable software package. PEEL\textsubscript{0} is a three-step workflow experiment as summarized in Figure 1.7. The first step prepares data by retrieving it from external websites and repositories. The second step executes the PEEL\textsubscript{0} model, which inputs, among other parameters, contain a classification scheme. The third step calculates areas at native resolution and aggregates the new classification to generate the final result.

To create a repeatable software package of the PEEL\textsubscript{0} experiment, the authors run the experiment within the PTU tool in the SOLE framework. By using Unix “ptrace” system call interposition, the PTU tool supports capturing a computational experiment using \textit{ptu-audit} and replaying an execution from a provenance trace using \textit{ptu-exec} (Figure 1.8). Once the PTU tool finishes capturing the experiment, it creates a package, which includes software binaries, dependent libraries, input data and a provenance trace of PEEL\textsubscript{0}. By sharing the software package, authors let reviewers and readers verify the PEEL\textsubscript{0} experiment using partial or full deterministic replay of the experiment.

The SOLE framework also provides a GUI interface for verifying a software package linked to a PDF-format paper. Readers and testers can easily browse the provenance graph of a referenced execution (Figure 1.9a) and re-execute the whole experiment or a process.
start experiment and audit

record events, copy files

ptu_exec
verify, start and redirect
read()
write()

PTU package

/bin/experiment

Figure 1.8: Using provenance to capture and re-execute in the PTU tool
(a) Explore information of a process
(b) Re-execute a process

Figure 1.9: Explore and re-execute a repeatable software package using an GUI interface in the SOLE framework selectively (Figure 1.9b).

1.6 Dissertation Roadmap

The chapters in this dissertation are organized as follows to present our work in greater detail.

Chapter 2: Related Work  surveys existing work on producing, presenting and reusing computational artifacts in scientific publications and describes how existing approaches do not meet all the goals mentioned above.

Chapter 3: SOLE Framework  We begin by determining if there is a single solution that solves the problem of making research papers reproducible. Our investigation, given the variety of computational artifacts associated with a science experiment, mandates a differing solution for verification and validation. That is there is no one-size-fits-all solution and reproducibility must be tackled at the different levels. Towards this end we propose the concept of science objects which consists of verification tools and sufficient amounts of metadata that must be present as part of these science objects to enable validation. We
describe a framework to create and maintain science objects, addressing **Goal G1**.

**Chapter 4: PTU** addresses the challenge of efficiently validating computational experiments at various granularity levels. We introduce our fundamental work on using Unix system call interposition for auditing provenance, which can be used to support efficient validation. This chapter addresses **Goal G2** of efficiency by allowing selective replay of the computational experiment, allowing for quicker validation of specific parts of the scientific experiment.

**Chapter 5: Network Provenance Capture and Replay with Minimal Setup** focuses on enabling reproducible research in many computational environments thus addressing **Goal G3** of generalizability. We propose different levels of network capture for different levels of verification and validation for network distributed experiments.

**Chapter 6: Closing Remarks** revisits the contributions of this thesis, draws conclusions, and provides future research directions in producing, presenting and reusing computational artifacts.
CHAPTER 2

RELATED WORK

Many solutions to the problems of producing and reusing computational artifacts have been studied over time. This chapter reviews the state of the art.

2.1 Computational Artifact Publication and Usage

There are different solutions to the problems of describing computational artifacts and using computational artifacts to support authors and readers in the conduct of reproducible research. These solutions can be categorized by where the artifact is placed and the dynamic nature of that artifact. Artifacts can be stored inside publications such as including source code within the published paper, or providing links to source code residing outside of the paper. Artifacts can be statically verified, by reading their corresponding source code, algorithm or data. Alternatively, artifacts can be dynamically verified, by re-executing programs and thus repeating experiments. Dynamic artifacts should be verifiable in the same setup from original publications or different setups, inputs, parameters, and environments. Table 2.1 highlights some existing solutions within those categories. We review and analyze each of those approaches with respect to the goals mentioned in Section 1.3 and our example in Section 1.5. Although each has interesting features, none can apply to our example or addresses all three of our goals.

Table 2.1: Classification of scientific publication computational artifacts

<table>
<thead>
<tr>
<th>Location of artifact</th>
<th>Verification approach</th>
<th>Example systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inside publication</td>
<td></td>
<td>Utopia</td>
</tr>
<tr>
<td>Static</td>
<td>Dynamic</td>
<td>Vistrails</td>
</tr>
<tr>
<td>Outside publication</td>
<td></td>
<td>Data-sharing solutions</td>
</tr>
<tr>
<td>Static</td>
<td>Code-sharing solutions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Combined solutions</td>
<td></td>
</tr>
<tr>
<td>Static</td>
<td>Software approaches</td>
<td></td>
</tr>
<tr>
<td></td>
<td>VM approaches</td>
<td></td>
</tr>
</tbody>
</table>
2.1.1 Inside Publication; Static Verification

One approach to providing verifiable results is to embedded code and data within published papers. This task can be done either manually by authors when they write the documents, or by using tools to automate the task. Utopia Documents [3] is an interactive PDF reader that associates concepts in paper with external annotations retrieved from an online metadata store. It provides access to information from various online data sources such as PubMed and Mendeley about specific terms or phrases selected by the viewer. Annotations are publicly shared and readers can comment upon them. Sweave [51, 52] and Dexy [62] are tools with code publishing features, which if adopted from the beginning of the scientific process can lead to papers with embedded source code and derived results.

These approaches either burden authors with manual mundane tasks or require them to adopt the tool early in their document writing process. Moreover, accessible source code and data are important but this approach is still not a silver bullet for verifying results. These approaches only provide a coarse level of verification, which does not satisfy G3 of verification with variable granularity. Our approach demonstrated in Section 1.5 does not require early adoption and can provide better means of verification with different levels of details.

2.1.2 Inside Publication; Dynamic Verification

Various workflow systems allow for the embedding of results in published papers. The Vistrails [5] workflow system associates figures and results in the paper with executable components. It allows authors to publish workflows and associated provenance and hyperlink to these artifacts in the article. It also provides provenance support for exploratory computational tasks that maintains detailed history information about the steps followed in the course of an exploratory task. GenePattern [82] is a genomic analysis platform that automatically captures the history of any computational work being done, allowing scientists
to generate pipelines to reproduce computational methods. It provides a Microsoft Word 2007 add-in [17] that allows scientists to embed their pipelines in a Word document. Using the add-in, readers of the document can rerun the pipelines on any GenePattern server from within the Word application.

Literate programming can also be used for dynamically assessing claims within a publication. An exemplar is the Sweave tool that allows an author to embed R code for complete data analyses in Latex documents. Readers and reviewers can assess the validity of the figures and plots in a publication by executing the Sweave file in their local R environment. Other similar works include Org-mode [84], Dexy and some closely related interactive-document approaches, IPython [67], Mathematica [95], and Sage [89] bind together code and the results generated by that code, which is clearly beneficial for readers and reviewers to verify the results.

While these examples serve as verifiable frameworks for reproducing research described in papers, authors must adopt such frameworks or platform prior to authoring papers. This requirement becomes an impediment for most authors, who often publish first and make publications reproducible post hoc, perhaps in response to their being highly cited by the wider research community. Hence, these examples do not provide for flexible adoption (Goal G2) and provide only a limited granularity level of verification (Goal G3). Our example in Section 1.5 shows that SOLE tools can be applied after the publishing process to augment the publication with language and data science object for static verification, or with a software package for dynamic verification and validation.

2.1.3 Outside Publication; Static Verification

Static-verifiable computational artifacts that reside outside scientific papers can be code and data. By sharing the code and data via web servers or code repositories, scientists can associate a large amount of information on code and data without worrying about page
limits publishers usually impose. This section discusses the data and code sharing approaches outside of publications.

**Data Sharing Approaches**

There are multiple platforms for sharing data. SEEK4science [96] is a web-based platform and associated tools that can be used for finding, sharing and exchanging data, models and processes in Systems Biology. Users can share data on the platform via sharing Excel spreadsheets, models and processes via a markup language specialized for the field. Dryad [33] is a repository of data for underlying scientific and medical publications. Using DOI, Dryad can reference to an entire set of data files associated with one publication or each individual data file within the data package. The ENCODE [15] project provides data for the entire human genome. All ENCODE data is available for visualization and data-mining using UCSC Genome and Table Browsers or via other provided tools from the project.

These platforms, while enabling easy data sharing, do not suffice for the validation of results in publications. However, they do provide a novel base for starting a verification process, which starts with retrieving data used by authors in their experiments.

**Code Sharing Approaches**

A common practice used by scientists to enable re-execution and replaying of experiments is to use open source software. The nature of open-source software makes it available to any readers of the publication or the users of the experiments. Open-source software can be obtained via multiple binary repositories, OS package managers, or via source code repositories such as GitHub [76] and SourceForce [39]. Program binaries can be downloaded and installed and used quickly. However, as discussed before, accessible source code and data are important but are not the all-in-one solution to the problem. Much free/open source software is poor or unusable [63]. Setting up the same environment as an original experiment can be time-consuming and error-prone. Moreover, although re-execution of the original experiment is possible, readers of the publication will not be able to easily isolate important
parts of the experiment of and can be attracted to unnecessary side-loaded details such as dependent library installation, data setup, runtime parameter configuration. Re-executing the experiment or part of it with different input files, parameters can be challenging and must be done manually at best. This state of affairs does not meet G1: to balance the trade-off between time and verification. On the other hand, section 1.5 shows that SOLE tools help authors to link “outside publication” science objects with research papers, hence remove the limitation while keeping the advantages of “outside publication” approaches.

2.1.4 Outside Publication; Dynamic Verification

Researchers often publish workflows to share experimental routines with colleagues for verification and reuse. These workflows can be created using a generic programming language, or constructed using a workflow application that provides a graphical user interface for composing pipelines based on a palette of computational tools. Several domain-specific online workflow repositories have evolved in recent years, such as BioCatalogue [10] and MyExperiment [31], as well as more general-purpose workflow management systems such as Galaxy [32], Kepler [1], Swift [99], Taverna [66], and VisTrails [46]. Using such scientific workflow development tools, scientists can design their workflows using available Web services. Once completed, these workflows can be published and shared in those dedicated repositories with colleagues, who can either use those workflows unchanged or compose new ones from existing service components. Since workflow systems allow data analyses to be recorded and subsequently used to reproduce results, they can play an important role in helping authors reduce time required to produce verifiable scientific papers, which meet our first goal of balancing between time and verification. However, similar to many dynamic verifiable inside-publication methods, these approaches require authors to embrace those frameworks or platforms prior to authoring papers, which conflicts with G2.

In a slightly different approach, some platforms allow sharing and running code. HUBzero
[56] is a platform that can be used to create web sites for scientific collaboration. HUBzero offers live published programs available for use instantly and entirely within an ordinary web browser. Instances of the HubZero framework for specific domains (nanoHUB [45], NEEShub [37], pharmaHUB [48]) allow analysis routines to be uploaded to a server, attached to simple graphical interfaces for passing parameters and viewing results, and executed remotely. RunMyCode [90] allows authors to upload code, going through policy and technical validation, then allows readers to re-execute the code. While these approaches are a step towards supporting validation, they do not satisfy goal G3 for variable levels of validation.

To conclude, we summarize our investigation of available tools in Figure 2.1. In a comparison with our demonstration in Section 1.5, repeatable software package science objects can be created in a post-hoc manner in multiple computational environments. This approach eliminates the requirement of adopting frameworks or platforms prior to authoring papers.
2.2 Computation Artifact Creation

In this section, we discuss software approaches and virtual machine (VM) approaches to creating computation artifacts. We also look at works that create and maintaining software pipelines.

2.2.1 Software Approaches

Different from above approaches, there are lower level approaches. Sumatra [21] is a tool for managing and tracking numerical computation projects in Python. CDE [36, 35] is a tool that uses system call interposition to automatically create portable software packages. CDE can create a package completely automatically, and run programs within a package without any installation. While these tools are generic to general experiments, they do not support different granularity levels of verification (G3). However, since CDE requires no configuration or installation, it is a promising approach that can satisfy our goals.

CDE: A Software Packaging Tool

The CDE tool aims to easily create a package on a source resource and execute a program in that package on a target resource without any installation, configuration, or privilege permissions. It runs in two main modes: audit mode to create a CDE package, and execution mode to execute a program in a CDE package.

In audit mode (Figure 2.2a), CDE uses UNIX ptrace system call interposition to identify
the code used by a running application (e.g., program binaries, libraries, scripts, data files, and environment variables), which it then records and combines to create a package. For example, when a process accesses a file or a library using the system call \texttt{fopen()}, CDE intercepts that syscall, extracts the file path parameter from the call, and makes a copy of the accessed file into a package directory, rooted at \texttt{cde-root} and consisting of all sub-directories and symbolic links of the original file’s location.

The resulting package can be redistributed and run on another target machine, provided that the other machine has the same architecture (e.g. x86).

In execution mode (Figure 2.2b), while executing a process from a package, CDE also monitors that process via \texttt{ptrace}. Each file system call is interrupted and its path argument is redirected to refer to the corresponding path of that file within the root directory of the CDE package on the target resource. In essence, CDE provides a lightweight virtualization environment to its running processes by providing the \texttt{cde-root} directory as a sandbox in a \texttt{chroot} operation. Redirecting all library dependency requests into this sandbox, CDE fools the target program into believing that it is executing on the original source machine [35]. It is to be noted that CDE binary only captures a single execution path, which is the execution path taken during run-time. If different execution paths need different types of dependencies, some dependencies may be left out. However, CDE does provide external scripts in its source code to find additional dependencies from strings inside binaries and libraries of captured packages.

\subsection{Virtual Machine Approach}

Some verification approaches are based on replay using a checkpoint mechanism. A checkpoint is created at regular intervals during program execution. At the beginning of each checkpoint interval, a snapshot of the process state is logged. This log includes information such as CPU registers, memory, and also kernel states corresponding to the process such as
file descriptor tables, signal handlers, etc. Then, using these logs, the state of that process at the start of each interval can be rebuilt by starting with the current state of that process and restoring the values from the log.

Revirt [23] and TTVM [44] enable re-execution by creating logs that contain input information from I/O calls and timestamps for the interrupts with the help of a virtual machine. Flight Data Recorder [97] uses hardware support to enable deterministic replay. It uses a copy-on-write hardware checkpointing scheme to create a log with I/O and interrupts data.

Kameleon [25] is user-friendly and can create bare bone virtual machine appliances in different formats for different Linux distributions. Users can use existing YAML-formatted [8] recipes or provide self-written recipes, macrosteps and microsteps to generate customized virtual images. Based on the recipe input, the Kameleon engine generates bash scripts to create an initial virtual image of a Linux distribution, and populates the initial image with more Linux packages to produce needed appliances.

Although these approaches have the advantages of highest level of granularity in replaying processes, they do not provide different granularity of verification. Using these approaches, users might be able to replay a whole experiment, but the ability to replay a fine-grained user-interest sub-process within the experiment is not easy to achieve. This leads to insufficient support for easy verification and repeatability of users which invalidates our goal G3.
CHAPTER 3
SOLE FRAMEWORK

In order to accomplish goals G1, G2 and G3, we have developed the SOLE framework [71, 74]. SOLE allows authors to present their publications in a descriptive and interactive manner to readers and reviewers. It consists of tools that aid authors in rendering their computational research in a form amenable to linking with a publication and for use by readers and reviewers for assessment and verification. SOLE addresses G1. By implementing and improving SOLE tools and in particular, the Provenance-To-Use (PTU) tool, we address G2. We broaden the applicability of PTU and other SOLE tools to address G3.

We next describe the SOLE framework and its elements, present its architectural components, and describe the tools that authors can use to create science objects and to present their publication in a descriptive and interactive manner.

3.1 Publication and Purpose
SOLE is interested in enabling the capture of research studies that process and combine large, diverse longitudinal, complex and distributed data using computing to generate results. We assume that such studies are primarily disseminated through conference and journal publications that typically have a page limit, thereby limiting descriptions in a publication. The investigators who perform such studies are the primary authors of these publications, which are assessed and verified by readers from the wider community.

Authors typically use the document metaphor in which descriptions and claims about the chosen scientific method are made with the help of textual elements, such as text, figures, tables, images, plots, and bibliographic links. A reader is typically interested in understanding the scientific method, but always reads the publication with a purpose, which may vary based on their need and role (See Table 1.1 from Section 1.2). For instance, some readers
may want to repeat experiments for reviewing purposes to verify the author’s claim that the experiments are correct. Alternatively, if an author claims development of a software package, readers may want to reuse that package in their own research.

3.2 Science Objects

The SOLE system provides authors with tools that turn data, methods, and processes used in a scientific method to granular units of knowledge, termed as science objects. Science objects are self-contained - they contain the content of a data, method or process and metadata, such as the Uniform Resource Identifier (URI) related to the content. The metadata and content of a science object may vary depending upon whether the object refers to a data, method or process. The objective is to use tools to build complete science objects so that they satisfy one or more of readers’ purposes as well as contain information that makes them amenable to be linked or embedded with a paper.

SOLE tools can be used to create two types of science objects: descriptive and interactive. Descriptive science objects are static and self-contained, in that the content in them is sufficient for a reader to satisfy their purpose. Such objects include annotations, source code fragments, provenance records, database records from a variety of sources, and software packages. Interactive science objects allow a reader to satisfy their purpose through execution of software. The objects have associated metadata that describe how to execute the software and are also aided with an accompanying infrastructure for execution. Examples of such objects are workflows and virtual machine images.

Prior to using a SOLE tool, the author must define the scope of the science object. This is achieved through a tag. Once a scope is defined, a SOLE tool can automatically populate the metadata and content of a science object. In general, to define the scope of a science object in SOLE, the author uses a tag with the following syntax:

\[ \text{begin type name}_1|\ldots|\text{name}_n \]
in which \textit{begin} and \textit{end} delimit the tag, \textit{type} defines the kind of science object to create, and \textit{name}_1 to \textit{name}_n are user-defined names. Thus, the same object can be tagged by more than one name. SOLE’s tools process the tag and based on each tag’s type definition creates a science object. The tools also associate a set of metadata elements representing the object, including a reference to the object as a URI. The exact location where the tag should be placed depends upon the type of science object. In the following list, the science objects in SOLE are described with where to place the tag and an overview of how the tools process the tags.

SOLE currently supports five kinds of science objects, which we now describe.

Language Objects

To create language objects, the author places the tags in source code files. SOLE uses Ctags \citep{26} to create the language objects in a file, and appends a URI to the language objects. We have expanded the Ctags utility to allow for user-defined tags as described above and to include more than one program function. This approach allows for linking an algorithm in a paper with multiple functions and data structures defined in multiple files.

Data Objects

To create data objects, the author places tags on PDF files or on the download history, if a dataset is downloaded from the Web. SOLE uses the Poppler library \citep{38} to extract tagged annotations from PDF. The metadata of the tagged annotation includes the URI of the PDF, the exact location in the PDF where annotation was made, and the annotated text. SOLE uses the Places schema of the Firefox browser to generate a linkable download history.
Software Package Objects

No tag is required to create a software package objects. The SOLE tool uses PTU for creating the package, but wraps the package with user-supplied tags and the resulting metadata annotated software package object is uploaded to the database (Chapter 4).

Workflow Objects

To create workflow objects, the author places tags on functions in the source file by inserting tags with the workflow tag type. Inputs, outputs and function definitions are determined through the tag type. These workflow objects are defined in Section 3.4.3. The SOLE tool creates a workflow object that conforms to the Galaxy-ES workflow systems. Galaxy-ES, based on the Galaxy system, provides an open, web-based platform for specifying tools and running computational experiments as workflows. Each tagged function is automatically wrapped as an appropriate Galaxy-ES tool definition and hosted on the web-server instance connected with Galaxy-ES. Authors can further specify if web services should accept user specified parameters and types of data.

In addition to the above described science objects, SOLE creates replayable packages (Chapter 4), which allow an author to assemble code, data, environment, and a reference execution trace into a single package that can be distributed easily, for example to testers who want to re-execute the full source code. Readers and testers can view the execution trace and specify, at a process and file level, nodes of the graph that they want to re-execute, thus saving time and effort for repeatability testing. This SOLE tool internally depends upon CDE, a utility for creating x86 Linux-based packages that requires no installation, configuration or root permissions. We have further extended the CDE utility to store a provenance graph at the process, and file-level representing a reference run-time execution, which the readers can view and replay.
3.3 The SOLE Environment

The SOLE environment hosts the publication and the science objects. The publication is rendered as a Web page; the science object specification is stored in a FluidInfo [80] database, a lightweight key-value data store, with a complete set of permissions to give users full control over their objects and tags. Depending on the type of science object, its content is either stored in the data store as a value, or the value is a URI that points to a file in the execution environment. Both the storage and the execution environment are currently on the Amazon cloud.

SOLE provides a simple query interface that allows users to search for science objects through their tag names. Since the same tag name may be associated with multiple science objects, the reader must first choose amongst the available options. SOLE provides the reader with a succinct description of the science object to allow the reader to make this choice. Once a science object is retrieved, the author chooses values, phrases, figures or tables within the publication content and links those chosen elements to the science object by clicking on “Attach SO”. Internally, this action links the chosen element with the science object in a manner similar to a citation being linked to a bibliographic entry. The environment supports some additional features, namely the bibliography specification of all science objects can be exported to CSV and XML. The authoring environment can be HTML or Word.

Figure 3.1 shows the SOLE architecture. Tools create and upload science objects to a SOLE environment, which consists of a metadata repository used to store information about science objects and an execution environment used for interactive science objects. The metadata repository acts as a web-based bibliography, indexed by tag names, which can be used by authors in their authoring environment to associate text phrases and figures with science objects.

Figure 3.2 shows the SOLE system in action through an example of a language object. The author first identifies the phrase e.g., “Aggregate” in the publication text that would
be better understood, assessed and/or verified by the reader if augmented. The author then tags the corresponding source code, runs a SOLE command line tool to create the necessary metadata, and finally associates the phrase “Aggregate” in the paper by linking it with its tag through a query-able interface. The reader can then view the implementation of the aggregate function and understand its merit.

3.4 Tools in the SOLE Reproducible Research Framework

In previous section, we have described the SOLE framework and its elements with its architectural components. We next use publication use cases from climate science, computer science, and biochemistry to illustrate science objects in the SOLE framework. Our climate science and computer science use cases employ publications by The Center for Robust Decision making on Climate and Energy Policy (RDCEP), a collaborative, multi-institutional project that aims to improve the computational models needed to evaluate climate and energy policies and to make robust decisions based on outcomes [81]. In RDCEP, sharing science objects in the form of data, tools, and software is critical; it enables scientists to compare models and to build more accurate models. Currently in RDCEP science objects
Figure 3.2: The steps that an author follows to create and associate science objects in SOLE.
are shared through a companion web site. We show here how to construct these science objects and link them with RDCEP publications using SOLE. We have also taken biochemistry publications [11] that use PubChem [92] datasets to demonstrate the use of annotated dataset science objects.

### 3.4.1 Language Object Creation Tool

Computational research papers abstract execution and source code and typically convey computational concepts through equations, methods, pseudo-code or algorithms. While such abstractions are necessary to communicate with the reader at a conceptual level, they can hide vital details that are present in the source code. For verification of the text, these details must be made available along with the conceptual description [59].

For instance, Cai et al. [12] describe a shape-preserving dynamic programming algorithm for multi-stage decision making problems. The paper describes the pseudo code for the numerical dynamic program but does not provide implementation details of the program, such as whether *floats* or *ints* are used for approximation or whether the program uses a top-down memoization approach or a bottom-up method for solving sub-problems. Such details if present along with the algorithmic description can be vital for a reader who plans to apply the algorithm on large amounts of data since a bottom-up approach has no recursion overhead.

Similarly interdisciplinary research papers on business and industrial engineering that use monte-carlo simulation methods to decide about risk and uncertainty often ignore finer details such as the chosen probability of distribution on inputs. In business domains, it is typical to use a triangular distribution instead of a uniform distribution, which is more common in industrial engineering. However, unless this finer detail is associated with the description of monte-carlo methods in the paper, it is difficult to quickly verify if the paper has adopted known knowledge.
SOLE’s language science objects improve overall readability and understanding of concepts described in a paper by annotating them with details present in source codes. Two approaches for annotation have been explored in previous work. In the first approach, the concept and the source code are mapped to a predefined ontology class. For instance, an initialization step and the configuration file will map to an initialization class. This approach is demonstrated in ontologies such as MGED [94] and EXPO [88] that allow for the linking of experimental descriptions with the resources used. In the second approach, there are no predefined vocabulary classes, but a language object is defined on the fly by taking a fragment from the source code and associating it with the algorithmic description in the paper. In SOLE, We follow the second approach since it allows greater flexibility.

**CTagsCrawler**  
SOLE uses the Ctags tool [26] and user-defined tags on source files to create language objects on the fly. By default, Ctags generates an index (or tag) file of language objects found in source files that allows these items to be located quickly and easily. A tag signifies a language object for which an index entry is available (or, alternatively, the index entry is created for that object). To identify a language object, Ctags identifies the language of the source code file and calls the appropriate language parser to operate on the file. Whenever the parser finds an interesting token, it calls a function to define a tag entry. Ctags parsers are available for more than forty programming languages. In SOLE, we have modified Ctags parsers for common programming languages, such as C, C++, JAVA, and R. The modified parsers identify both language specific tokens and user-defined tokens (described shortly), which can be present within a language object, span across more than one language token, or across source code files.

The user-defined tokens are declared according to the following syntax:

```
### t@ name_1 | ... | name_n
language science object content
### t@
```
This syntax is similar to the syntax in Section 3.2, in which the words begin and end are replaced with the comment style of the source code, type defines the type of source code object, and the name declares the arbitrary concept to which the user wants to associate with the language object(s). The name is indexed and tagged, and the author can use the tag to associate this user-defined object with the phrases in the text. Figure 3.3 shows an R function that is annotated with user-defined tags. It also shows the accompanying metadata about the function, which is part of the language object.

Demonstration  Best et al.’s “Synthesis of a Complete Land Use/Land Cover Dataset for the Conterminous United States” [9] describes a method for generating a land cover dataset for the conterminous USA called “PEEL0.” The paper proposes a novel synthesis methodology, which combines information from multiple sources by establishing a common classification scheme at lower spatial resolution. It shows how to generate the dataset by combining data from existing data products, including the MODIS Land Cover Type (MLCT) and the National Land Cover Database (NLCD).

In the paper, the synthesis methodology, which is a 3-step process, is described conceptually. The first step of the methodology is shown in Figure 3.3b. Finer algorithmic details, such as the specific R functions used for re-projection or reclassification are not described. By attaching a language object, the author can make the definition of these terms precise and remove ambiguities in interpretation. Annotated science objects from this paper are demonstrated in the SOLE system [70] and linked to specific text phrases.

3.4.2 Data Object Creation Tools

Research papers commonly cite data through a bibliographic reference or through a digital object identifier (DOI). The latter approach provides a permanent identifier for the data, even if the location of data or its metadata change over time. While these citation mechanisms
are necessary for linking datasets they may be insufficient for the purpose of repeatability since they do not specify which part of a dataset is used in the research paper. For instance simply referencing a report that is 100 pages long often does not help. A reviewer verifying values or tables taken from the report must know the references at the granularity of pages and paragraphs.

The problem of coarse linking of datasets is acute when the web link refers to an online database such as PubChem. This database allows users to access subsets of data (e.g., molecules) from its repository through Web forms and query interfaces. Accessed datasets have an important scientific value for research, but are commonly cited using the web address of the primary repository. Research papers may provide text descriptions of how the dataset was downloaded from the primary online database, and include specification of query parameters, such as dates or spatial region clauses. In a few cases publications provide a link to the downloaded dataset. In neither case can the reviewer verify how the dataset was derived from the database, with the specification often falling short. Reviewers often have to re-learn the mechanism for downloading the dataset as used in the publication. It would be helpful if the downloaded dataset could be identified with a link or a DOI that is derived from the primary database’s link or DOI and that also includes metadata, such as query

![Source Code](attachment:image.png)

(a) A language object with tagged source code  
(b) Conceptual Description in the Paper

Figure 3.3: Enhancing conceptual description with Language Science Objects
parameters that describe how the dataset was obtained.

An important direction towards citing data at a finer granularity is the Open Archives Initiative Object Reuse and Exchange (OAI-ORE) project [50, 49], which exposes the rich content in Web resources to support their reuse in authoring, exchange, and preservation. Web resources are considered as compound digital objects consisting of many distributed resources with multiple media types including text, images, data, and video. The project goal is to expose the rich content in these aggregations through a specification, which hopefully may become a standard for describing Web resources.

While OAI-ORE provides a useful specification for Web resource citation, the need to supply many metadata values as part of the specification can be cumbersome for an author. Therefore in SOLE our primary focus is on tools that can provide input values to the metadata specification in an automated way. SOLE provides two tools for automating the metadata extraction problem. For PDF sources, SOLE provides, PDFExtractor, a tool that extracts segments from a PDF or postscript document and produces a bibliography of the segments. The segments have the DOI of the primary PDF but are enhanced with additional metadata as required by the OAI-ORE specification. If the data source is an online database, SOLE provides RepAudit, a browser plugin to silently audit the parameter specification or the link traversal. It provides both the downloaded dataset in the original format and also provides a metadata file, based on OAI-ORE specification with values of the metadata entries. We use examples to illustrate the use of the two tools:

PDFExtractor PDFExtractor extracts segments, such as text, images that are selected by the user through the annotation command in the PDF reader. Figure 3.4 shows a PDF document with a segment selected by the user. Figure 3.5 shows the resulting science object, indexed through user-defined tags. Metadata in the resulting science object is according to the OAI-ORE specification and enables precise location of the segment within the PDF. Internally the PDFExtractor uses the Poppler library [38] to extract the segments and attach
Figure 3.4: Annotating a PDF by selecting a rectangular region and adding tag Texas metadata specification to the extractions. If the user is connected to a SOLE environment, PDFExtractor also transmits the bibliographic information to the FluidInfo database.

**Demonstration** We asked the authors of “Feasibility of U.S. renewable portfolio standards under cost caps and case study for Illinois” [41] to annotate their paper with pointers to referenced data and code. This policy paper surveys US state renewable portfolio standards (RPS) with cost caps, and studies in particular the feasibility of the RPS proposed for Illinois through a cost model. The paper reports RPS-related data from more than a dozen reports (cited in the references) published by different federal and state energy departments. Often RPS and cost values, for purposes of brevity, are aggregated from their original source and an average is mentioned. It is difficult to verify such a policy paper unless the papers are referenced with page numbers and the approximate location of each value on the page.

We used SOLE to annotate the reports cited in this paper [69]. For example, the reference “[13]” mentioned in the excerpt of the paper in Figure 3.5 is one report that is tagged. The tagging is shown in Figure 3.4. The authors simply annotate the PDF with an easy to remember tag, such as “Texas”, and then execute the PDFExtractor tool on the saved PDF. The tool generates a corresponding science object with the additional metadata about the page number in the PDF and approximate location on the page, and also uploads the science objects to the SOLE system. The author then attaches these science objects to the text phrases in the paper. The reader can then verify that the authors have used an approximate value of $1 per MWh in the new paper while the referenced paper stated “average of less than $1/MWh”.
RPS was initially met by existing hydropower rather than new construction [6]. Texas (1999) possesses such anomalously strong wind resources that development of windpower in the state could be driven largely by the federal Production Tax Credit and Investment Tax Credits with the state RPS playing a much less significant role. (The Production Tax Credit and Investment Tax Credits, henceforth “PTC”, reimburse qualifying renewable generators for up to 50% of the installed cost. See Appendix A.1 for further discussion.) Current Texas REC prices remain so low (∼ $1 per MWh [13]) that the state RPS is not a significant subsidy for windpower in Texas, and current construction implies that Texas wind capacity will reach its 10 GW target almost fifteen years ahead of RPS-mandated requirements [14]. Prediction of the expected evolution of renewables implementation

Figure 3.5: The extracted text from the paper (top). The highlighted value $1 per MWh refers to the annotation science object with the tag Texas (bottom)
RepAudit  RepAudit is a browser plugin for Mozilla Firefox. If enabled by the user before exploring an online database, RepAudit first determines the primary link of the online database being explored. If no link can be determined, RepAudit assigns a temporary link, which can be changed later by the author. RepAudit audits user’s links and/or the keywords mentioned in query forms and maintain a history of the visited links and pages in an SQLite database. This database is built using the Firefox Places schema, which stores new URLs and their pages. The RepAudit database also stores link traversal history. The history for a downloaded dataset can be exported as a derived link.

Demonstration  We have experimented with the use of RepAudit for various biochemistry publications that use data from PubChem. These publications are provided by PubChem as dataset exemplars. To locate the correct data set as described in the publications, we worked with biochemistry graduate students at our institution. Here we describe the use of RepAudit on one of those publications.

The biochemistry paper “Identifying Compound-Target Associations by Combining Bioactivity Profile Similarity Search and Public Databases Mining” [13] provides the following dataset description:

The NCI-60 data set contains anticancer screening results for more than 40,000 compounds. It is publicly available in the PubChem BioAssay database (38) as 73 bioassays with the name of NCI human tumor cell line growth inhibition assay under the DTP/NCI data source. In this work, only the top 60 bioassays (referred hereafter as NCI-60) with the largest number of tested compounds were selected (Supporting Information, Table S1). Relevant bioactivity data were downloaded at the PubChem FTP site (ftp://ftp.ncbi.nlm.nih.gov/pubchem/Bioassay, accessed on December 9, 2010). A total of 5083 compounds were found commonly tested in all of the 60 bioassays. Additional data set characteristics are summarized in Supporting Information.

In collaboration with graduate biology students we searched PubChem for the NCI-60
The NCI-60 data set contains anticancer screening results for more than 10,000 compounds. It is publicly available in the PubChem BioAssay database as 72 biosamples with a name of "NCI human tumor cell line growth inhibition assay" under the "DTP/NCI data source." In this work, only the top 60 biosamples (selected hereafter as NCI-60) with the largest number of tested compounds were selected (Table S1). Relevant bioactivity data were downloaded from the PubChem FTP site (ftp://ftp.ncbi.nlm.nih.gov/pubchem/Bioassay) accessed on December 9, 2010. A total of 1,083 compounds were found commonly tested in all of the 60 biosamples. The bioactivity profile of each compound was derived by extracting the log(10) value obtained from the NCI-60 cell lines, where G140 is the concentration required for the 50% growth inhibition of tumor cells. 631 compounds with missing log(10) value in one or more of the NCI-60 cell lines were discarded. Additionally, 150 compounds were further discarded, because they exhibited identical bioactivity in all NCI-60 cell lines, which made them less informative and unsuitable for bioactivity profile similarity calculation (see below). A total of 426 compounds were collected and used for constructing the bioactivity profile database. The original bioactivity profile data for these compounds are available in Supporting Information, Table S2. Additional data set characteristics are summarized in Supporting Information, Figure S1 with respect to six physicochemical properties: molecular weight, octanol-water partition coefficient, number of hydrogen bond acceptors, number of hydrogen bond donors, and topological polar surface area.

BioActivity Profile Similarity Search (BASS)

The BASS approach consists of three major steps (Figure 3.1). For a given query compound in the NCI-60 data set, we first searched against the entire bioactivity profile database and calculated pairwise bioactivity profile similarity for each reference compound in the data set and the query compound. Second, a neighbor compound was identified if its bioactivity profile similarity is above a selected threshold. Finally, the known target of the query compound is predicted as the potential target of its neighbor compound or vice versa. A critical step of BASS is to identify the neighbor compounds for a given query compound based on the similarity of bioactivity profiles, which is defined as Pearson correlation coefficient ($r_B$),

$$r_B = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum(x_i - \bar{x})^2 \sum(y_i - \bar{y})^2}}$$

where $i$ equals 0, and $x_i$ and $y_i$ are the log(10) values of the $i$th NCI-60 cell line for compound $x$ and compound $y$, respectively. In this work, $y$ is considered as a neighbor compound of $x$ if $r_B$ is above 0.75. This similarity threshold was selected based on a statistical test, which was carried out by randomly selecting two compounds from the entire bioactivity profile database for 100,000 times and recording each time the bioactivity profile similarity. A probability ($p$-value) was subsequently calculated for obtaining a bioactivity profile similarity above a certain threshold. For the similarity threshold of 0.75, a $p$-value of 7.36e-11 was found to be a good balance.

Figure 3.6: A browser history science object shows paths and keywords used to search for data set NCI data set. Even though the description mentions NCI-60 as publicly available from PubChem site, no assays can be retrieved with the specific search term. However, 195 bioassays are indeed retrieved from the search term “NCI human tumor cell line growth inhibition assay” with 75 bioassays corresponding to bioassays under the source DTP/NCI uploaded earlier than December 9, 2010. There is no way of sorting or filtering the bioassays on the largest number of tested compounds from the website from both search and advanced search features. In addition the URL ftp://ftp.ncbi.nlm.nih.gov/pubchem/Bioassay provides no search interface or a directory of NCI-60 dataset. Thus it is not entirely evident what specific search terms were used by the authors and the filters applied by them.

In Figure 3.6, we show the PubChem dataset description tagged with the browser history of the downloaded dataset. The history is shown in three representations: (1) as a full path from the beginning of the session to the downloaded dataset in the end (Figure 3.7), (2) as...
Figure 3.7: First 10 links of the path from the beginning of the browser session to the downloaded dataset step in the end

distinct links in the path (Figure 3.8), and (3) as search terms (Figure 3.9). The obtained history shows that downloading a dataset requires significant data retrieval from PubChem and that a simple URL is not sufficient for obtaining the exact same data.

3.4.3 Workflow Object Creation Tool

The production of a typical computational paper may involve numerous distinct data management and computational tasks, such as gathering and preparing input data; configuring; building and running the model on various computing resources; collecting metadata about the model; computing environment and input files; post-processing model outputs; and visualization. The paper mostly describes the end results of these tasks. However, a reader may want to change one or more parameters or steps in this process to assess the impact(s) on the result. This task can be made easier by allowing the reader to change parameters via command-line interfaces, without having to change source codes and recompiling the application. However, substituting one or more steps with another is often more challenging.
Distinct Path

1. The PubChem Project  
2. NCI human tumor cell line growth inhibition assay - PubChem BioAssay - NCBI  
3. AID 93 - PubChem BioAssay Summary  
5. No items found - PubChem BioAssay - NCBI  
9. No items found - PubChem BioAssay - NCBI  
10. No items found - PubChem BioAssay - NCBI  
11. No items found - PubChem BioAssay - NCBI  
13. No items found - PubChem BioAssay - NCBI  
15. No Title  
16. AID 93 - PubChem BioAssay Summary  
17. No items found - PubChem BioAssay - NCBI  
18. AID 3 - PubChem BioAssay Summary  
19. AID 3 - PubChem BioActivity Analysis: Data Table  
20. PubChem BioActivity Analysis: Data Table  
21. PubChem BioActivity Analysis: Data Table  
22. 417470910365453968.csv.bz2

Figure 3.8: Distinct link used in the browser session
Distinct Search Terms

1. "NCI human tumor cell line growth inhibition assay" [Assay Description]
2. NCI human tumor cell line growth inhibition assay
3. (#1 AND "DTP/NCI"[Source Name]
4. (NCI human tumor cell line growth inhibition assay) AND "DTP/NCI"[Source Name]
5. "NCI human tumor cell line growth inhibition assay" [Assay Name]
6. "NCI human tumor cell line growth inhibition assay" [Assay Comment]
8. NCI human tumor cell line growth inhibition assay. Data for the NCI/ADR-RES Breast cell line [Confirmatory]
9. NCI human tumor cell line growth inhibition assay. Data for the NCI/ADR-RES Breast cell line
10. "NCI human tumor cell line growth inhibition assay. Data for the NCI/ADR-RES Breast cell line"[Assay Name]
11. "NCI human tumor cell line growth inhibition assay. Data for the NCI/ADR-RES Breast"[Assay Name]

Figure 3.9: Search terms used in the browser session
without source code modification.

Workflow systems provide abstractions for designing, composing, and executing applications with multiple steps such that steps may be conveniently substituted and parameters easily modified. When the task is supposed to be performed many times, such abstractions make it easier for the scientist to focus on the logic of their applications instead of the technical details for accessing and invoking the software components needed for execution, which can be delegated to the workflow system. Despite the advantages, most domain-specific workflow systems are often strongly typed necessitating a scientist to abandon their favorite programming environment.

Galaxy-ES is a workflow system for climate science that allows a scientist to remain with his or her programming environment but provides support for wrapping their application with simple, uniform Web interfaces. An application can be executed as multiple steps, while the system automatically manages the computational details. From these Web interfaces a reader can change parameters to run an application. Galaxy-ES builds from the Galaxy system in genomics, which allows experimentalists without informatics or programming expertise to perform complex large-scale analysis with just a Web browser. Galaxy-ES, similar to Galaxy, also introduces two other features for reproducible research: (1) it automatically tracks and manages data provenance and provides support for capturing the context and intent of computational methods, and (2) Galaxy Pages are interactive, web-based documents that provide users with a medium to communicate a complete computational analysis.

However, to use Galaxy-ES the author still has to provision an instance and build a tool definition of their application. In particular, to provision and define a tool an author has to write three files: (1) a configuration file describing the tool user interface and how the tool has to be executed including inputs and outputs; (2) a launcher script that prepares the custom execution environment and redirects output and error streams to a self-describing container; and (3) wrap scripts that perform the input dataset staging, the actual processing
and the output dataset staging. These steps require familiarity with the internals of the Galaxy system and the details associated with tool definitions, which may be significant for an author.

To reduce this gap, we extended the SOLE approach of tagging source codes. Authors continue to work in their favorite programming language and tag functions in source codes with the tag definition as described in Section 3.2. The tag_type describes whether the tagged portion of the code describes inputs to a function, outputs of a function, or the function definition. SOLEConnect is a SOLE tool that interprets these tag types and creates corresponding Galaxy-ES tool definitions, which are then automatically deployed on a Galaxy instance running on Amazon cloud. SOLEConnect also creates a corresponding science object of this tool describing metadata elements such as the URL of the tool, its inputs, outputs and the corresponding language science object. The metadata elements are uploaded to FluidInfo.

3.4.4 Discussion

Research ideas are often non-trivial and complex. Evaluating a non-trivial idea completely is often beyond the time budget of any single paper as this requires running many benchmarks, models, competing solutions. Consequently research publications explore novel ideas but without the associated rigor of experimentation. Given increased competition in research fields, we believe that lack of rigor will continue, creating an author-reader divide where an author presents a biased evaluation but a reader cannot completely verify the merit of the idea.

In SOLE, we have aimed to minimize this divide by creating a layer of tools and interfaces which allow authors to share their computational artifacts and readers to verify them through interfaces. For the greater part, the system does assume that authors and readers have the necessary time and incentives to share and verify. Repeatability testing for conferences
presents an interesting case in this dimension where authors and readers have sufficient incentives through the conference system, but limited time to share and verify. We believe the both SOLE tools and particularly PTU tool are important research contributions in that direction although for lack of measures it is difficult to quantify the cost of reproducibility.

An additional concern is determining which papers to choose to reproduce. In SOLE, we have assumed that this decision is made by a mutual consensus between co-authors and readers. In general, aggregating a universal consensus on which publications must be reproduced is hard. However, we do believe that funding agencies can play a pivotal role in this decision since they are the primary sponsors of the research and also a proxy for the interests of the general public.

Finally, in SOLE our focus has been towards a general approach to reproducibility. However, making a publication reproducible has indeed required us to understand concepts in climate science and biochemistry. Minimizing the involvement of the human-in-the-loop is one of our goals. We believe that we will have a better understanding of this challenging problem as publications, software, and tools from other disciplines become accessible to use and we can use SOLE tools to make them reproducible.
CHAPTER 4

PTU

The second goal of SOLE framework is to minimize authors’ effort in creating reproducible science objects and prolong these objects so that they can be accessed and executed on-demand by users to validate or repeat an experiment in a publication.

As introduced in section 1.1 and 1.2, reproducing experiments from scientific papers presents multiple challenges to both authors and readers. Making experiments reproducible has minimal incentives, and making scientists adapt to new complex computational environment is time consuming. However, we are seeing an increasingly phenomenon that conference committee and journal editors are encouraging or requiring authors to submit their code, data and software for repeatability testing [91]. Repeatability testing improves peer review in three ways: (1) It allows reviewers to not only read the ideas in the paper, but validate them by running the accompanying software; (2) It allows reviewers to work the software for different kinds of data and parameters to check robustness of the idea; and finally (3) It allows reviewers to determine the limitations and assumptions in the ideas by checking for boundary conditions.

In this chapter, we will introduce the PTU tool in Section 4.1 as a utility for packaging the software programs and its reference execution without modifying the application. The tool aims to make it easy and attractive for authors to use it, and a fine control, efficient way for testers to do selective repeatability testing. We then extend PTU in Section 4.2 to include software provenance as part of a software package. We will show how this provenance information can be used to build and maintain software pipelines. Finally, in Section 4.3, we introduce a front-end interface for users to view provenance from software package created by PTU, allows users to explore provenance data associated with each process and data, to understand their data and library dependencies, as well as to re-produce the result via re-execution. This interface also allow authors of research papers to link texts in their PDF-
4.1 PTU: Provenance-To-Use

4.1.1 Motivation

As documented by recent experiments, repeatability testing can be arduous and time consuming for both authors and testers [28, 55]. Authors have to prepare code, document it, and make explicit rendering of all dependencies on compilers, operating systems and hardware. For testers, assessing code and data for repeatability can be challenging since documentation is rarely complete and perfect. But more so, as experiments become data and computation intensive, the testing time can be significant [83].

Recently some tools have emerged that aid authors and testers in making their software and thus experiments repeatable. CDE helps authors to package the code, data, and environment for Linux programs so that they can be run and deployed on other machines without any installation or configuration [35, 36]. The resulting x86 Linux-based package requires no installation, configuration, or root permissions. Being open-source, with a user-friendly interface for packaging and rerunning, CDE has become useful for conducting reproducible research.

While CDE is a step towards simplifying repeatability testing for authors, it does not reduce computation and data processing time for repeatability tests. Testers often want to selectively repeat portions of an experiment that may be of interest without having to go through the time-consuming process of repeating the experiment in its entirety. For instance, testers may choose to avoid running a compute-intensive process that decodes and splits an MPEG-video, or avoid performing a data-intensive text scan, or avoid network communication or transfers during repeatability testing of an experiment. Similarly, they may choose to reuse cached results from the past to run the experiment. The ability to
selectively run a program depends upon the granularity at which the program has been audited and captured in a previous reference run.

### 4.1.2 PTU: Provenance-To-Use

Provenance-To-Use (PTU) [68, 72] is a SOLE tool for reducing time for repeatability testing. Authors can use PTU to accomplish two tasks: (1) build a package of their source code, data, and environment variables, and (2) store process and file level details about a reference execution of their system in an accompanying database. A package alleviates testers from software deployment issues allowing for convenient distribution. Preserving a reference execution path and run-time details within a package eases distribution of this vital information that can guide testers during the testing phase. In particular, testers can explore the provenance graph and accompanying run-time details to specify the part of the provenance graph that they want to re-execute or replay. Figures 4.1, 4.2, and 4.3 show the modes of operation.

PTU uses CDE [35, 36] to create and run a package. CDE uses Unix `ptrace` system call interposition to collect code, data files, and environment variables. The `ptrace` mechanism
Figure 4.2: Tester: views the provenance graph stored in the database of the PTU package

Figure 4.3: ptu-exec restarts from /bin/calculate without re-executing /bin/workflow, uses CDE to re-route file dependencies
also allows for auditing file and process information, which can be transformed for storing a provenance graph, independent of the application. We enhance the ptrace mechanism in CDE to store a provenance graph representing a reference run-time execution at the process and file level. We describe how authors and testers can use PTU.

**For Authors:** To create a self-contained package with a reference execution, the author prepends the application command with the `ptu-audit` tool as in the following example, which involves the Java application TextAnalyzer applied to a file news.txt:

```
ptu-audit java TextAnalyzer news.txt
```

The `ptu-audit` tool internally uses `ptrace` to monitor about 50 system calls including process system calls, such as `execve()`, `sys.fork()`; file system calls, such as `read()`, `write()`, `sys.io()` for collecting file provenance; and network calls, such as `bind()`, `connect()`, `socket()`, and `send()` for auditing network activity. Whenever system calls occur, PTU notes the identifier of the process that made the system call to extract more information about the process from the Linux `/proc` file system. In particular, the following information is obtained: process name, owner, group, parent, host, creation time, command line, environment variables, and a file’s name, path, host, size, and modification time. A separate thread is used to obtain memory footprint, CPU and I/O consumption of the process from `/proc/$pid/stat` every three seconds.

In the case of distributed applications involving multiple compute nodes, processes are audited independently at each node using `ptrace`, i.e., without coordination with the other node. Provenance information about processes and files is stored in the form of an Open Provenance Model (OPM) [58] compliant provenance graph in a SQLite database. Currently, the provenance collector in PTU makes two assumptions. First, if multiple cooperating processes modifying files concurrently, cycles may appear. The provenance collector in PTU does not ensure that cycles and duplicates do not appear in the provenance trace. Second, only network connection information is audited and no network dumps are made. Hence,
testers cannot replay a network distributed computation without actually conducting network communication.

When a system call (file or network) returns, and if a new file or network does not exist, PTU emulates CDE functionality. In the case of a file, it copies the accessed file into a package directory that consists of all sub-directories and symbolic links to the original file’s location. In the case of a network it copies the network connection information, ordered by time under the directory named by the IP and port information. The package directory also contains the SQLite database that stores the provenance information of the test run. When the entire test run finishes, PTU builds a reference execution file consisting of the topological sort of the provenance graph. The nodes of the graph enumerate run-time details such as process memory consumption, and file sizes. The tester, as described next, can utilize the database and the graph for efficient re-execution.

For Testers: Prior to running a package a tester can view the provenance graph of the accompanying reference execution. This graph can be viewed at the granularity of processes and files, and can aid the tester in visually determining parts of the program that they wish to re-execute. For processes, an accompanying bar graph shows CPU and memory consumption. To specify a re-execution, a tester can either specify nodes on the provenance graph or modify a run configuration file that is included in the package. The configuration file initially specifies the provenance graph, corresponding to the reference execution, ordered topologically. A tester can turn flags on or off for each process and files in the provenance graph to specify if the process needs to be run or if the file needs to be re-generated.

To run the package again, testers prepend the program command with a ptu-exec tool as:

```
  ptu-exec java TextAnalyzer news.txt
```

The ptu-exec tool uses the provenance graph/run configuration file to determine if any additional process(es) must be run or file(s) must be re-generated. A re-run of a process or
a regeneration of a file is mandatory if:

1. A process/file is in the descendant sub-graph of another process that is marked for re-running.

2. A process/file is in the descendant sub-graph of another file that is marked for regeneration.

Re-running these additional processes and regenerating the files is necessary to maintain consistency of the provenance that will be obtained from the test run. To re-execute, *ptu-exec* obtains run configuration and environment variables for each process from the SQLite database. To re-execute a process, *ptu-exec* again monitors it via *ptrace* and re-executes CDE functionality of replacing path argument(s) to refer to the corresponding path within the package *cde-package/cde-root*/. By doing so, *cde-exec* creates a chroot-like sandbox that fools the target program into believing that it is executing on the original machine [35].

*ptu-exec* offers other command-line options for testers to control testing. A few primary command-line options are listed here:

- `-time -t1 <t1> -t2 <t2>` implies the tester can re-execute everything between `t1` and `t2`. If `t1` is null or `t2` is null, execution is done from beginning to `t2`, or from `t1` till end. Parameters `t1` and `t2` are specified as the number of seconds since the Epoch (1970-01-01 00:00:00 UTC). With this option the runtime configuration file does not need to be modified.

- `-input <p> <f1> <f2>` allows user to specify input `f1` instead of `f2` for process specified with process id `<p>`. This helps user test the workability of the system.

To avoid becoming a bottleneck due to file copy operations, PTU uses CDE to provides users with a fine control of how files are handled. For example, users do not want to include some input data files that are large and stored in a network mounted directory, users can
set PTU’s options to not copy those files to the PTU package. Users can decide to ignore certain file names or files in certain directories. These options are defined in a configuration file using the following syntax:

- `ignore_prefix={all file paths having this prefix are ignored}`
- `ignore_exact={an exact file name to be ignored}`
- `ignore_substr={all file paths having this substring are ignored}`

These options can be used multiple times within a configuration file, as in Figure 4.4.

A key distinguishing characteristic of our approach is that it can provide fast re-execution and correct output even when no workflow definition or management system is in place. PTU does not force testers to keep mental track of processes that were run, while allowing them to use simple command-line options. Our design does assume that provenance metadata is complete and does not omit, alter, or create false provenance events, and is stored persistently.

### 4.1.3 Use Case and Experiments

To test PTU we took two papers [9, 61] in which authors shared data, tools and software. We constructed meaningful testing scenarios and determined if PTU provides any performance improvements. The first paper is from Best et al. that describes the synthesis methodology to generate a new land cover dataset for the conterminous USA called PEEL. The methodology is implemented as an R-program and combines information from multiple sources by establishing a common classification scheme at lower spatial resolution. This is a three-step workflow process, in which the second step is to execute a model which inputs among other parameters a classification scheme. This step is memory-intensive and a typical run takes close to a GB of real memory. To reduce memory consumption, testers may want to further lower the spatial resolution by specifying even fewer input files and a simpler classification model.
# These directories often contain pseudo-files that shouldn’t be tracked
ignore_prefix=/dev/
ignore_exact=/dev
ignore_prefix=/proc/
ignore_exact=/proc
ignore_prefix=/sys/
ignore_exact=/sys
ignore_prefix=/var/cache/
ignore_prefix=/var/lock/
ignore_prefix=/var/log/
ignore_prefix=/var/run/
#ignore_prefix=/var/tmp/
ignore_prefix=/tmp/
ignore_exact=/tmp

# un-comment the entries below if you think they might help your app:
#ignore_exact=/etc/ld.so.cache
#ignore_exact=/etc/ld.so.preload
#ignore_exact=/etc/ld.so.nohwcap
# Ignore .Xauthority to allow X Windows programs to work
ignore_substr=.Xauthority

# Ignore so that networking can work properly
ignore_exact=/etc/resolv.conf
# These files might be useful to ignore along with /etc/resolv.conf
# (un-comment if you want to try them)
#ignore_exact=/etc/host.conf
#ignore_exact=/etc/hosts
#ignore Exact=/etc/nsswitch.conf
#ignore_exact=/etc/gai.conf

# Access the target machine’s password files:
# (some programs like texmacs need these lines to be commented-out,
# since they try to use home directory paths within the passwd file,
# and those paths might not exist within the package.)
ignore_prefix=/etc/passwd
ignore_prefix=/etc/shadow

Figure 4.4: Example of using PTU configuration file to control file inclusion
Our second program is a text analysis program that mines newspaper articles obtained freely via the Internet to ask questions about water management issues. The program is based on the Java-based Unstructured Information Management Architecture (UIMA) and runs a named-entity recognition analysis program using several data dictionaries. The process splits the input file into multiple input files and runs a parallel analysis on all input files. The tester may want to rerun the program on one input split, but with higher convergence criteria to manually verify the correctness of the program. Note that in general testing these programs will require R and UIMA to be installed, but by using CDE, testers are able to test without any prerequisites.

The graphs generated at the process and file level in both programs are big. PEEL0 generates a provenance graph with five process nodes, and 10000 exclusive file reads based on the number of files in the dataset, and 422 exclusive file writes for the aggregated dataset. In TextAnalyzer, a single run of the job generates a provenance graph of eight process nodes, total conducting 616 exclusive file reads, 124 exclusive file writes, and 50 file nodes that are read and written again. To improve the readability of these graphs, a single process view is shown that shows just the sub-graph corresponding to that process – the files that it reads and the files that it writes. The tester notes the process identifiers of the start process for replay, which in case of both PEEL0 and TextAnalyzer remains as the first process. The tester also writes a configuration file in which different inputs are specified for selected processes: in the case of PEEL0 input to the first process is a changed directory location pointing to the new set of files, and for the second process it is the location of the file specifying a simplified classification scheme; in the case of TextAnalyzer it is a set of process identifiers to re-execute with a different parameter value. Figure 4.6 and 4.5 show the provenance graph and configuration file for the PEEL0 program.

Figure 4.7 shows the performance improvements of using PTU with PEEL0, and Figure 4.8 shows the performance improvement of using PTU with TextAnalyzer. The most impor-
Figure 4.5: PTU configuration file to control different inputs

Figure 4.6: Provenance graph of PEELd program
Tant statistic is the slow down in running the process during a test execution at about 35% for PEEL₀ and about 15% for TextAnalyzer. This slowdown is due to additional system call tracing, with many file system calls traced for PEEL₀. Both these times are easily offset during the re-execution phase. TextAnalyzer has a significant improvement (>98%) since the entire process is run on a much smaller file.

4.1.4 Summary

PTU is a step toward testing software programs that are submitted to conference proceedings and journals to conduct repeatability tests. Peer reviewers often must review these programs in a short period. By providing one utility for packaging the software programs and its reference execution without modifying the application, we have made it easy and attractive for authors to use it and a fine control, efficient way for testers to use PTU.
4.2 Auditing and Maintaining Provenance in Software Packages

“... releasing shoddy VMs is easy to do, but it doesn’t help you learn how to do a better job of reproducibility along the way. Releasing software pipelines, however crappy, is on the path towards better reproducibility.” C. Tituss Brown 1

In previous section, we have introduced PTU, a valuable tool to make provenance-aware software package for repeatability testing. PTU, similar to other software approaches reviewed in section 2.2 that ensure reproducibility, has a limitation that it does not provide support for combining and merging multiple shared packages for a software pipeline. This conflicts with our second and third goals stated in Section 1.4 for an efficient reproducible research process for authors while making it easy for readers to conduct research reproducibility at various granularity levels in different computational environments. In this chapter, we will introduce PTU-SP as an extension to PTU to include software provenance as part of a software package. We will show how this provenance information can be used to build and maintain software pipelines.

1. SOURCE http://ivory.idyll.org/blog/vms-considered-harmful.html

Figure 4.8: Time reduction in testing TextAnalyzer using PTU
4.2.1 Introduction

Computational reproducibility is a challenge, yet crucial for science. To meet the challenge, large-scale science projects are increasingly adhering to reproducibility guidelines. For instance, software associated with a publication is made available for download (see Figshare [87], RunMyCode [90], and Research Compendia [85]); but increasingly many science projects are making end-to-end software pipelines available. These pipelines are often for the larger scientific community, as in the case of Bio-Linux 5.0 [47], which is a bioinformatics virtual machine that provides access to several pipelines for conducting next-generation sequence analysis, or sometimes to demonstrate project impacts as in the case of Swift Appliance [64], a virtual machine, which demonstrates crop simulation models using workflow systems.

To help projects adhere to these reproducibility guidelines, project members often adopt best practices and tools for developing and maintaining software so that their contributed software quickly becomes part of a pipeline. In this chapter, we focus on software packaging tools. We describe how auditing and maintaining software provenance as part of a packaging tool can significantly help in building and deploying software pipelines. In particular, provenance can be helpful in cutting down manual effort involved in ensuring software compatibility, thus leading to improved administration of software pipelines.

A software pipeline consists of many individual software modules. Given the collaborative nature of science, it is not uncommon for modules to develop independently. Furthermore, a module itself may depend upon externally-developed libraries, which evolve independently. To ensure library compatibility, and avoid what is often called "dependency hell", a software module is often packaged together with specific versions of libraries that are known to work with it. In this way, contributing project members can ensure that their module will run on any target system regardless of the particular versions of library components that the target system might already have installed.

However, packaging software modules with associated dependencies, but without clearly
identifying the origin of the dependencies, gives rise to a number of provenance-related questions, especially when constructing software pipelines. For instance, determining the environment under which a dependency was built or other dependencies which must be present for using a module, are questions that must be answered when combining packages for creating software pipelines. Similarly, if a new software package is released, then through dependency analysis it will be useful to know which packages of a pipeline can use it. If a new version of a library is released that contains security fixes, then it will be useful to know which pipelines or packages are vulnerable.

To answer such questions, we must be able to capture and determine the provenance of a software entity, i.e., capture and determine where it came from. Current package management systems do not provide a means to audit or maintain software provenance within it. We use CDE, a software packaging tool that creates a source code and data package while identifying all static and dynamic software dependencies. CDE has also been successfully shown to create software packages out of many development environments. Though CDE packages static and dynamic dependencies for an application, it does not store associated provenance.

The first contribution of this section is to enhance PTU to include software provenance, i.e., provenance of shared libraries and binaries on which a program depends. We call this enhanced PTU as PTU-SP. We describe tools and methods to audit, store, and query this provenance in PTU-SP. We then describe a science project use case in which software reproducibility is a concern. Our second contribution is to show how provenance, audited and stored as part of a PTU-SP package, can help in creating software pipelines for this use case. Finally, we show how provenance can be maintained as new packages are built during construction of software pipelines.

The remainder of the chapter is structured as follows: In Section 4.2.2, we describe provenance that can be audited, stored, and queried in PTU-SP, resulting in a provenance-
included package. In Section 4.2.3 we describe a science use case where provenance, included as part of software packages, can help in creating pipelines. In Section 4.2.4 we further enhance PTU-SP to enable it to maintain correct provenance as new packages are created. In Section 4.2.5, we conduct a thorough experimental evaluation to measure the overheads associated with auditing and maintaining provenance. We conclude in Section 4.2.7.

4.2.2 PTU-SP: Software Provenance in PTU

The objective of auditing provenance is to capture additional details of the creation and origins of a library or a binary, such as the version of the compiler, the compilation options used, the exact set of libraries used for linking. This information must be gathered on a per environment basis so that it becomes easy to compile and create software pipelines.

Audit CDE’s audit feature identifies static and dynamic program dependencies. We instrument this feature to first determine a dependency tree, and then use UNIX utilities to store additional provenance information about each dependency. To create a dependency tree, process system calls are monitored that audit process name, owner, group, parent, host, creation time, command line, environment variables and the process binary’s path. Whenever a process executes a file system call, a dependency of that process is recorded. In general, this dependency can be a data file or a shared library. We identify shared libraries using standard extensions, such as .so for system libraries and .jar for Java libraries, and create a dependency tree based on these libraries. Information about binaries and required shared libraries, such as version number, released version of shared libraries, and associated kernel distribution, is audited using UNIX commands file, ldd, strings, and objdump. By including these commands, we can obtain other static and dynamic dependencies, some of which are not audited by CDE during run-time. This set of commands is a more comprehensive way of obtaining dependencies compared to CDE’s external scripts. Current operating system distribution and user information is recorded from command uname -a and function
Table 4.1: LevelDB key-value pairs that store file and process provenance. Capital letter words are arguments.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid.PID1.exec.TIME</td>
<td>PID2</td>
<td>PID1 wasTriggeredBy PID2</td>
</tr>
<tr>
<td>pid.PID.[path, pwd, args]</td>
<td>VALUES</td>
<td>Other properties of PID</td>
</tr>
<tr>
<td>io.PID.action.IO.TIME</td>
<td>FILE(PATH)</td>
<td>PID wasGeneratedBy/wasUsedBy FILE</td>
</tr>
<tr>
<td>meta.agent</td>
<td>USERNAME</td>
<td>User information</td>
</tr>
<tr>
<td>meta.machine</td>
<td>OSNAME</td>
<td>operating system distribution</td>
</tr>
</tbody>
</table>

getpwuid(getuid()).

Storage Each package can store captured provenance in a relational database. Since this provenance will be useful for whatever target resource package is being used, we believe it is best to store this provenance within the package itself. We use LevelDB, a very fast and lightweight key-value storage library for storing provenance. To store provenance graphs that contain process-file and process-process edges, in a key-value store, we encode in the key the UNIX process identifier along with spawn time. The value is the file path or the process time. Table 4.1 describes the LevelDB schema for storing provenance graphs:

Query LevelDB has a minimal API for querying. Instead of providing a rich provenance query interface, currently we implement a simple, lightweight query interface. The interface takes as input the program whose dependencies need to be retrieved. Using a depth first search algorithm, a dependency tree in which the input program is the root is determined. The result is saved as a GraphViz file. Since the result may include multiple appearances of common files like those in /lib/, /usr/lib/, /usr/share/, and /etc/ directories, the query interface also provides an exclusion option to remove uninteresting dependencies.

4.2.3 Using PTU-SP Packages to Create Software Pipelines

We describe a software pipeline through a use case. We then describe how PTU-SP packages can help to create the described software pipeline. The use case will also be used for experimental evaluation in Section 4.2.5.
Software Pipelines

Scientists with varying expertise at the Center for Robust Decision Making on Climate and Energy Policy (RDCEP) engage in open-source software development at their individual institutions, and rely primarily on Linux/MacOS X environments. The Center often needs to merge its individual software modules to create software pipelines. We describe software modules being developed by three scientists, henceforth denoted as Alice, Bob, and Charlie, and the associated software pipeline that needs to be constructed.

- **A** measures and characterizes land usage and changes within it. She develops **data integration methods** to produce higher-resolution datasets depicting inferred land use over time. To develop the needed methods, her software environment consists of R, geo-based R libraries (raster, ggplot2, xtable, etc.), and specific versions of Linux packages (r-base v2.15, libgdal v1.10, libproj v4.8).

- **B** develops **computational models** for climate change impact analysis. He conducts model-based comparative analysis, and his software environment consists of A’s software modules to produce high-resolution datasets, and other Linux packages, including C++, Java, AMPL [27] modeling toolkits and libraries.

- **C** uses A and B’s software modules within **data-intensive computing methods** to run them in parallel. C’s scientific focus is the efficiency of distributed computing methods and his software environment is primarily Java and Python and their libraries on Linux.

- For the **Center**, the goal of their combined collaboration is to predict future yields of staple agricultural commodities given changes in the climate; changes that are expected to drive, and be influenced by, changes in land usage [24]. The Center curator’s environment is Mac OS X and a basic Unix shell.
Given the linear workflow of the science problem, it is often the case that B needs to rerun A’s software in his own environment. Instead of installing, this task can simply be achieved if A shares a CDE package with B. However, if B attempts to create a software pipeline that includes A’s package and her software modules, then he needs to verify the provenance of each dependency included in A and her software. This issue arises a dependency with the same file path, but built on different Linux distributions (therefore different content), will conflict. In fact, if B creates a CDE package corresponding to this pipeline, one of the dependencies will be overwritten in the newly created package. By using the provenance-enabled CDE packages, which store md5 checksums of dependencies, such origins can be immediately verified, without manually tracking kernel distributions on which the dependency was built or communicating with the author of the software. Similarly, by checking versions of all dependencies within the package, B can document the compatibility of the newly created software pipeline.

As the use case demonstrates, C needs to use A’s and B’s packages, and the problem of dependency tracking, i.e, determining distributions and versions, given several dependencies and software environments, can increase significantly. Their specified software environments, which may appear different, can be still overlapping. To demonstrate the magnitude of overlap, we assume that each developer uses the cloud for their research, which is not uncommon in today’s projects, and chooses a different Linux distribution. Differences in the choice of linux distributions is also not surprising as the Linux Counter Distributions Report [53] indicates that there is no clean winner in terms of usage of Linux distributions, with no one

Figure 4.9: Software Packages of A, B, and C
distribution accounting for more than 30%. Further, we limit software environments to refer to application binaries and libraries that are often overlapping and create conflicts.

If the two assumptions are sound, then the overlap in the environment, i.e., files which have the same path, but differing content, can be as high as 15%. We calculate this figure by taking five Linux distributions with similar setup available on Amazon EC2. For each pair of machines, we calculate the number of files with the same path on two machines, and the number of files with the same path on two machines but having different md5 checksum. Table 4.2 shows that between any two machines, on average, 6.8% of files have the same path but differ in content. In other words, these files are not interchangeable but depend on the underlying operating system.

### Table 4.2: Ratio of different files having the same path in 5 popular AMIs. The denominator is number of files having the same path in two distributions, and the numerator is the number of files with the same path but different md5 checksum. Omitted are manual pages in /usr/share/ directory.

<table>
<thead>
<tr>
<th></th>
<th>Red Hat Linux</th>
<th>SUSE Linux</th>
<th>Ubuntu 12</th>
<th>Ubuntu 13</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon AMI</td>
<td>5498 / 23k</td>
<td>3184 / 11k</td>
<td>1203 / 5.4k</td>
<td>1819 / 5.5k</td>
</tr>
<tr>
<td></td>
<td>(10.5% / 44.5%)</td>
<td>(6.3% / 22.2%)</td>
<td>(2.5% / 11.3%)</td>
<td>(3.3% / 10.0%)</td>
</tr>
<tr>
<td>Red Hat</td>
<td>3861 / 12k</td>
<td>1654 / 6.6k</td>
<td>2223 / 6.3k</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(7.7% / 23.6%)</td>
<td>(3.4% / 13.7%)</td>
<td>(4.0% / 11.5%)</td>
<td></td>
</tr>
<tr>
<td>SUSE Linux</td>
<td></td>
<td>1245 / 3.9k</td>
<td>2085 / 6.4k</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(2.6% / 8.2%)</td>
<td>(3.8% / 11.7%)</td>
<td></td>
</tr>
<tr>
<td>Ubuntu 12</td>
<td></td>
<td></td>
<td></td>
<td>8226 / 24k</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(15.0% / 43.0%)</td>
</tr>
<tr>
<td>Number of files</td>
<td>52.5k</td>
<td>50k</td>
<td>48k</td>
<td>55k</td>
</tr>
</tbody>
</table>

#### 4.2.4 Merging Provenance in PTU-SP

While provenance-included packages can eliminate much of the manual and tedious efforts of ensuring software compatibility, the downside is that provenance stores within a package need to be effectively maintained as software pipelines are themselves cast into new packages. Consider the Center’s need for creating a software pipeline that satisfies reproducibility
guidelines. To help the Center build this software pipeline, assume A, B, and C share their individual provenance-included packages. By exploring A, B and C’s package provenance, the Center can examine all data and control dependencies among the contributing packages. The Center can then define a new experiment with steps using data and control dependencies from the three contributed packages, and create a new software package of this experiment. In particular, correct pathnames, attribution, etc., will need to be verified. We next describe how PTU-SP, with a \(-m\) option, can be used to merge provenance from contributing packages.

In the typical CDE audit phase, file system binaries and libraries found in the path of program execution are copied to the \textit{cde-root} directory. However, provenance may indicate two dependencies with the same path but emerging from different distributions or versions. In PTU-SP, these two files are stored in separate directories identified by a UUID, which is unique to the machine on which PTU-SP is executed. The UUID is the hash of the Mac address and the operating system. By creating this separate directory based on a UUID, files with the same paths but different origins can be maintained separately. Note that only files with differing content but the same path are maintained in separate UUID directories. Files with different paths can all still be in the same generic \textit{cde-root} folder. We also include versioning of UUID directories so that they are copied and maintained correctly in new packages.

Because provenance requires that separate UUID based directories be created within a PTU-SP package, correspondingly, modifications are needed in the LevelDB provenance store and the PTU-SP redirection mechanism. The LevelDB path in the value field needs to reflect the UUID directory where the dependency exists. The CDE redirection, which redirects all system calls to the \textit{cde-root} directory, in PTU-SP needs to redirect to the appropriate UUID directory. This redirection can be tricky since it needs to know where the process is running. To enable correct redirection, PTU-SP with merge maintains a \textit{current_root_id} pointer for
Figure 4.10: A’s package is reused in C’s package. PTU generates a subgraph “$f_4 \rightarrow Aggregation \rightarrow f_5 \rightarrow Generate\ Image \rightarrow f_6$” from C’s package. Since that subgraph is subgraph isomorphic to the provenance graph in A’s package, PTU can validate A’s authorship on a part of C’s package.

each tracing process. This bookkeeping pointer helps in redirecting to the package root directory of the pointer in case the process forks other processes. Alternatively, if the process performs an `execve()` system call, or accesses a file, or changes directories, absolute paths are read and checked to determine if redirection is necessary.

Another issue when merging two packages is maintaining licensing information. While general licensing issues are outside the scope of this chapter, the current PTU-SP maintains authorship of software modules during the merge process. When two packages are merged in their entirety, the authorship of a new package is the combined authorship of the contributing packages. However, when part of a contributing package is used to create a new package, then authorship must be validated from the provenance stored in the original package. To validate, PTU-SP generates the subgraph associated with the part of the package, and, using subgraph isomorphism, validates that it is indeed part of the original provenance graph (Figure 4.10).

The subgraph isomorphism (or matching) problem is NP-complete [93] leading to an ex-
ponential time algorithm. In our case, we compare file paths and names to determine if two provenance graphs are subgraph-isomorphic. In our implementation of the VF2 subgraph-isomorphism algorithm [16], we reduce computation time by only matching provenance nodes of processes with the same path to their binary and working directory, and only matching provenance nodes of files with the same path. We believe that this implementation is sufficient for validating provenance subgraph isomorphism among lightweight packaging tools.

4.2.5 Experiments and Evaluation

The benefits of reproducibility can be hard to measure. In this Section, we describe the three experiments we conducted to determine the overall performance of PTU-SP.

1. We determined the performance of PTU-SP in: auditing performance overhead, disk storage increase, and provenance query runtime;

2. We determined the redirection overhead if multiple UUID-based directories are created in PTU-SP; and

3. We compared the lightweight virtualization approach of PTU-SP with Kameleon [25], a heavyweight virtualization approach used for reproducibility.

All experiments in this section are tested on an Ubuntu 12.04.3 LTS workstation with an 8GBs RAM and 8-core Intel(R) processor clocking at 1600MHz.

Audit Performance and Size Overhead in PTU-SP

In table 4.3, we record execution times and disk usage of CDE and PTU-SP in auditing a software pipeline mentioned in section 4.2.3. Both PTU-SP and CDE are set up for a pipeline with two applications: Aggregation and Generate Image. Each is repeated 10 times. The result shows approximately a 2.1% slowdown of PTU-SP in comparison with CDE
Table 4.3: Increase in PTU-SP performance is negligible in comparison with CDE

<table>
<thead>
<tr>
<th></th>
<th>Create Package</th>
<th>Execution</th>
<th>Disk Usage</th>
<th>Provenance Query</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDE</td>
<td>852.6 ± 2.4 (s)</td>
<td>568.8 ± 2.4 (s)</td>
<td>732MB</td>
<td></td>
</tr>
<tr>
<td>PTU-SP</td>
<td>870.5 ± 2.5 (s)</td>
<td>569.5 ± 1.8 (s)</td>
<td>732MB + 236kB</td>
<td>0.4 ± 0.03 (s)</td>
</tr>
</tbody>
</table>

due to provenance capture. The result fits with our observation that the overhead is from `ptrace` which both CDE and PTU-SP rely on heavily to implement their capture capabilities. Additional functions that store provenance record to LevelDB database introduce negligible provenance capture overhead compared to 0-30% CDE virtualization overhead [35]. In this setup, CDE package uses 732MB; while PTU-SP, in addition to the software package, creates a LevelDB database of size 236kB (0.03% increase) that contains approximately 12,000 key-value pairs.

To measure provenance query performance, we created a Python script to query the audited LevelDB provenance database and create a provenance graph of the experiment with common shared libraries filtered out. The Python script reads through approximately the 12,000 key-value pairs in 0.39 seconds to create a GraphViz script that can be converted to image or visualized later.

Redirection Overhead in PTU-SP

We also compared an execution of CDE package and PTU-SP package to measure the redirection overhead of PTU-SP. Using the packages created by the above experiment with two applications, `Aggregation` and `Generate Image`, we pipelined output of `Aggregation` to input of `Generate Image`, which requires PTU-SP to apply redirection among multiple CDE roots. The experiment showed 3 data files, as outputs of `Aggregation` package, were moved to `Generate Image` package. After the data was moved to the next package, the experiment was executed the same as in CDE. The result shows less than a 1% slowdown of PTU-SP, which maybe due to initial loading of library dependencies in `Generate Image` package.
PTU-SP Vs Kameleon

In this experiment, we used the Kameleon engine to make a bare bone VM appliance that contains the content of a PTU-SP package corresponding to the software pipeline described in the use case (Section 4.2.3). The package content was copied directly to the root file system of the VM appliance. In terms of user software, the new VM appliance is close to a replica of the package, without any redundant installed software. We compared the two approaches qualitatively and quantitatively.

Qualitatively, the overhead of instantiating a VM is significant as compared to creating a PTU-SP package. In particular, for PTU-SP the user needs to specify input packages, and using one command, the author can create a new software package. Kameleon is user friendly and can create virtual machine appliances in different formats for different Linux distributions. But, users must provide self-written YAML-formatted [8] recipes or self-written macrosteps and microsteps to generate customized virtual images. Based on the recipe input, Kameleon generates bash scripts to create an initial virtual image of a Linux distribution, and populates the initial image with more Linux packages to produce needed appliances.

Quantitatively, we compared the time for executing the software pipeline within a PTU-SP package with time for execution within a VM. Note that we do not compare time for initializing, since time for writing YAML scripts cannot be measured in the case of Kameleon. During the execution, PTU-SP redirected 2717 file-read system calls, 10 file-write system
calls, 17 file-read-write system calls. Figure 4.11 shows that the Kameleon VM appliance slowed down the experiment significantly: approximately 200% or more. This heavyweight VM overhead is substantial in comparison with the PTU-SP lightweight approach.

4.2.6 Related Work

Details about software have been included in provenance collected within workflow systems. For instance, Research Objects [7], packages scientific workflows with auxiliary information about workflows, including provenance information and metadata, such as the authors, the version. Our focus here is not limited to any specific workflow system.

Software packaging tools such as CDE [35, 36] and Sumatra [21] can capture an execution environment in a lightweight fashion. Sumatra captures the environment at the programming level (Python), while CDE operates at the operating system level, and is thus more generic. Even at the system level, different tracing mechanisms can be used. At the user-space level, ptrace [78] is a common mechanism, whereas at the kernel-level, use of SystemTap [75] is more common. SystemTap, being kernel-based, has better performance compared to ptrace since it avoids context switching between the tracee (which is in the kernel) and the tracer (which is user space) [42]. However, from a reproducibility standpoint, SystemTap needs to run at a higher privilege level, i.e., it requires root access, creating a more restricted environment.

Virtual machine images (VMIs) provide a means of capturing the environment in a form that permits later replay of a computation. Kameleon [25] uses a bash script generator to create virtual images from scratch for any Linux distributions. Using recipes, users can generate customized virtual images with predefined software packages to run on different cloud computing service providers. We have compared our approach with creating VMIs for reproducibility.

Tools such as ReproZip [14] have demonstrated the advantages of including provenance
in self-contained software packages. Currently, these tools include execution provenance and not software provenance. Finally, software provenance is an emerging area that uses Bertillonage metrics for finding software entities in large code repositories [20]. In this section, we have described how software provenance can help in building packages that can satisfy reproducibility guidelines.

### 4.2.7 Conclusion

CDE is a software packaging tool that helps to encapsulate static and dynamic dependencies and environments associated with an application. However, CDE does not encapsulate provenance of the associated dependencies such as their build, version, compiler, and distribution. The lack of information about the origins of dependencies in a software package creates issues when constructing software pipelines from packages. In this section, we have introduced PTU-SP, which can include software provenance as part of a software package. We have demonstrated how this provenance information can be used to build software pipelines. Finally, we have described how the PTU-SP can maintain provenance when used to construct software pipelines.

### 4.3 Provenance-Aware Electronic Publication

Previous sections 4.1 and 4.2 have introduced the command-line version of PTU and its extension. In this section, we will introduce a front-end interface for users. This GUI interface allows users to view provenance from software packages created by PTU, to explore provenance data associated with each process and in/output, to understand their data and library dependencies, as well as to reproduce the result via re-execution. Additionally, we have implemented a PDF Reader interface and a Latex style class that allows authors to link texts in a PDF file to corresponding provenance artifacts.
4.3.1 Introduction

As mentioned in chapter 1, encoding a computation-based experiment in a text-based paper is nearly impractical. Therefore, current research papers summarize the associated data and computation rather than reproduce it computationally. To support their research claims, research papers currently adopt indirect means, such as building companion websites that share data and source code. However, these external websites remain disconnected from the content within the paper, making it difficult to verify claims and reproduce results. This disconnect not only affects readability of a computational publication but limits readers and reviewers from being able to assess the validity of findings, determine workability of results on different parameters, or reuse data and methods for their own research.

This section, using SOLE with PTU as a back-end framework to create computational artifacts, describes a simple linkage method to create, display, and replay those computational artifacts within a PDF-format electronic paper. In this method, we provided a Latex style class with two customized Latex commands and a PDF & provenance reader. Using these supplied tools, authors can create a provenance-enabled PTU package and publish a PDF-format electronic paper that links to the package. Readers can view the paper on any normal PDF reader software such as Adobe Acrobat Reader (Windows OS), Preview (Mac OS), Evince or Okular (UNIX OS). However, provenance-related features of the PDF-format paper will not show on these PDF readers. Instead, users will be informed and directed to use our SOLE PDF Reader.

The contribution of this section is to provide a simple novel tool for linking between static texts in PDF-format file and processes and data in experiments. We call this SOLE PDF Reader. We describe the tools and methods to audit and create a provenance-aware package. We provide methods to embed provenance artifacts from provenance-aware packages into Latex source to create a PDF file, and to view the PDF file and related provenance artifacts in our SOLE PDF Reader. Finally, we show how the provenance artifact linkage can help
to re-execute and validate processes from within our SOLE PDF Reader.

The remainder of the section is structured as follows: In subsection 4.3.2, we describe provenance that can be audited, stored, and queried in PTU resulting in a provenance-included package. In subsection 4.3.3, we show a science use case where provenance included as part of software packages can be embedded into a Latex document and viewed in SOLE PDF Reader. In subsection 4.3.4, we further show how to use SOLE PDF Reader to re-execute processes recorded in the provenance-included package. In subsection 4.3.5, we conduct an experimental evaluation to measure the performance of the SOLE PDF Reader with a large package and how easily the SOLE PDF Reader can be used. We conclude in subsection 4.3.6.

4.3.2 Provenance Discovery

Provenance discovery has been presented in detail in Sections 4.1.2 and 4.2.2. In this section, we explain how to explore and query provenance.

There are several methods to explore the captured provenance in a PTU package database. Users can dump the database to a text file, create an OPM-compliant provenance graph to view, or write their own programs using APIs provided in Python, C, C++, etc. for LevelDB.

PTU provides Python scripts to explore its provenance database. Using “printdb.py”, users can dump all contents of PTU’s database. This method can be useful for debugging with UNIX grep utility. Using “db2dot.py”, users can generate an OPM-compliant provenance graph where processes and files are shown and linked with “used”, “wasGeneratedBy”, “wasTriggeredBy” relationships. This simple visualization tool provides a quick way to verify the provenance of data in an experiment. The provenance graph is filtered to remove common files to avoid clustering and unnecessary viewing obstacles. PTU also has a script to generate a timeline that shows the creation time of all provenance artifacts.

While these PTU scripts can help to explore the captured provenance in an easy-to-see
visualization, they do not provide flexibility in these types of visualization. In particular, detailed provenance information such as process start time, stop time, CPU usage, and memory usage, etc. are not shown in such graph visualization. In fact, adding information to graphs can make the graphs too crowded for rendering and viewing purposes. A more flexible and interactive interface is needed to allow users to explore fully PTU’s captured provenance.

Faced with the challenge of providing a flexible and interactive tool for provenance exploration and re-execution, we introduce SOLE PDF Reader with three main features:

1. **To provide linkage from content of scientific papers to their corresponding provenance artifacts of any computation-based experiments** This linkage can facilitate provenance exploration from context of the paper that describes the experiment. This linkage can also support connections of external resources to data within the experiment by the means of description provided by authors in their papers.

2. **To provide linkage among provenance artifacts within an experiment** By showing dependencies among processes and files within the experiment, SOLE PDF Reader allow users to explore the dependency from input data to final results of the experiment. Visualization of the provenance can help to understand execution paths taken within the experiment.

3. **To provide detailed information of each provenance artifact within an experiment** These details include static information such as a binary path, current working directory, and dynamic runtime information such as input parameters, start/stop times, and machine resource usages. By providing these details, the tool can support reproducing results by re-executing processes with recorded parameters using PTU capability.
Figure 4.12: SOLE PDF Reader interface for provenance exploration: an experiment run two processes “process.py” in two remote machines, used scp to copy their outputs back to local machine, then execute “summary.py” to generate the final output.

4.3.3 Provenance Linkage and Presentation

SOLE PDF Reader supports provenance linkage, exploration and re-execution via three components: a Provenance Explorer interface, a PDF Reader interface and a document linkage module. In this section, we present these interfaces with their implementations.

Provenance Explorer Interface

Figure 4.12 shows a provenance graph of an experiment. Processes are presented as rectangles whereas files are portrayed as eclipses. Network sockets are displayed as eclipses similar to files. Arrows present different meanings as specified in Table 4.4. Users can explore a process by clicking on that process’ rectangle to view additional information on the left panel, or in the Stats tab of the graph panel. The left panel shows its parent process id, process id, binary path, initial working directory, input parameters, start and end times of that particular execution, cpu and memory usage. Similarly, clicking on a file’s eclipse reveals its associated path. And a socket’ eclipse shows its source and destination IP addresses.
Table 4.4: Provenance edge in SOLE PDF Reader’s Provenance Explorer interface

<table>
<thead>
<tr>
<th></th>
<th>Process P2</th>
<th>File F2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process P1</td>
<td>P1 → P2: P2 wasTriggeredBy P1</td>
<td>P1 → F2: F2 wasGeneratedBy P1</td>
</tr>
<tr>
<td>File F1</td>
<td>F1 → P2: P2 used F1</td>
<td>not exists</td>
</tr>
</tbody>
</table>

PDF Reader Interface

With a provenance-aware PDF-format paper, readers can open the PDF file in SOLE PDF Reader to view the paper and explore associated provenance. The PDF file is rendered in SOLE PDF Reader using PDF.js JavaScript library [2] in a WebKit module of QT Framework [77]. Inside SOLE PDF Reader, the PDF file is rendered and provided a view similar to common PDF readers. However, if the PDF file was prepared using SOLE tools to associate with provenance, some text in the PDF file would be linked to provenance artifacts. When users click on such text, the corresponding provenance artifact will be shown on the Provenance Explorer interface as in Figure 4.13.

To apply this feature, the authors use Latex to create the paper and follow three steps to make the PDF-format paper provenance-aware:

1. The authors include a Latex style class “sole.sty” (downloaded from SOLE PDF Reader website) in their Latex source.

\usepackage{sole}

2. The authors upload their provenance archive tar-ball to their website and add a SOLE link at the beginning of their Latex file, following the syntax:

\soleinit{http://website/path_to_hosting_directory}{filename.tgz}

where the first parameter is the online directory and second parameter is the tar ball filename.
3. The authors link text from the paper to its corresponding provenance artifact by using the artifact identifier obtained from the Provenance Explorer interface in the format:

\sole[dbid=#dbid,type=#type,part=#part]{#artifactid}{displayed text}

The parameters in the square bracket are optional. Using this command in a Latex source will create a hyper-link in its target PDF file using the following format

http://base_url/solelink?dbid/#dbid/type/#typeid/part/#partid/id/#artifactid

This link will be parsed and intercepted by SOLE PDF Reader to show the linked provenance artifact. If the file is opened by different PDF reader software, the linked text will appear as normal hypertext. That hypertext links to a web page asking users to use SOLE PDF Reader instead.
4.3.4 Process Re-execution Interface

While PTU has the capability of re-executing processes captured in PTU packages, there is a disconnect from the PDF-format paper to the process re-execution. In a common scenario, readers have to go through a series of steps to re-execute a process in PTU package:

1. Read the experiment description in the paper
2. Match that description to a process node in a provenance graph
3. Change their current working directory to the corresponding working directory of that process node
4. Provide the identity of that process node to PTU-exec for a re-execution

The Provenance Explorer interface and PDF Reader interface described above have eliminated the second step, but not the last two steps. This seriously breaks the streamline of reading and validating experiments in papers. Hence, we have added a Re-Execution interface to the application.

From the Provenance Explorer interface, SOLE PDF Reader provides a link to the Re-Execution interface whenever readers investigate a process node. The Re-Execution interface allows readers to re-execute the process described in that process node with the same original command-line parameters, environment, and its current working directory. The interface also lets readers run that process with different parameters, which can help to facilitate the validation process of claims from research papers. Figure 4.14 shows a process “summary.py” being re-executed.

4.3.5 Experiments and Evaluation

We have evaluated SOLE PDF Reader on three provided interfaces: the PDF Reader interface, the Provenance Explorer interface, and the Re-Execution interface. The experiments
are run on a machine with Intel(R) Core(TM) i7-3770 CPU, 8 cores @ 3.40GHz, 8GB RAM, running Ubuntu 12.04.3 LTS.

**PDF Reader interface performance** The PDF Reader interface has been evaluated with PDF-format papers with number of pages ranged from 4 - 100 pages: a TAPP13 paper (4 pages) [72], an IPAW14 paper (12 pages) [73], and a Ph.D. thesis (100 pages). We measured PDF loading time as the time from the PDF file loaded to the time the first page is rendered and viewable. In our experiment, PDF loading time was less than 500ms in each case, which gave acceptable user experience. We also measured preparation overhead of using sole.sty style file in Latex source with “pdflatex” command. In our experiment with
the three mentioned papers, there was no noticeable preparation overhead for using `sole.sty` Latex style file for linking text with provenance artifact.

**Provenance Explorer interface**  The Provenance Explorer was evaluated with provenance databases of different number of artifacts. Our experiments showed that the graph render time was less than 500ms for up to 1000 of processes, files and sockets. The provenance graph became unreadable when there were more than 1000 artifacts drawn together; hence we limited the number of displayed artifacts to 1000. When the Provenance Explorer focused on a single process, the total number of dependencies was reduced, hence the view was clearer and became readable to users.

**Re-Execution interface**  Since Re-Execution interface is a wrapper around PTU-exec command, there is no overhead in using the Re-Execution interface. Several users’ feedbacks showed great interests in using this interface to quickly and automatically determine and define a re-execution of a process. We have not been able to get significant feedback for our SOLE PDF Reader. However, the current feedback indicates that users are satisfied with its usefulness, performance and user-friendliness.

### 4.3.6 Summary

PTU is a utility for packaging software programs and its reference execution without modifying the application. This section introduces SOLE PDF Reader as a connection from PTU packages to research papers. SOLE PDF Reader allows authors to link texts in a PDF file to corresponding provenance artifacts in a PTU package. We also demonstrate SOLE PDF Reader in facilitating paper readers to view provenance, to explore provenance data associated with each process and its data, and to re-produce the result via re-execution.

By providing PTU as a backend and SOLE PDF Reader as a front-end of the SOLE framework, we have accomplished goal G2 to “increase the efficiency of reproducible re-
search process for authors, while making it easy for readers to conduct verification at various granularity levels.”
CHAPTER 5
NETWORK PROVENANCE CAPTURE AND REPLAY WITH MINIMAL SETUP

In chapter 4, PTU was introduced to use UNIX “ptrace” system call interposition to capture experiments on a single machine. However, the third goal G3 of this thesis aims to provide authors with tools to conduct reproducible research in multiple computational environments. In this chapter, we present a new capability of PTU: capturing computation provenance from experiments running in network distributed environments independently of any workflow systems with minimal setup. By capturing provenance of network data flows at different level of granularity, PTU supports from provenance query to offline replay of network communication. We show that the tool is efficient with low level of overhead and can provide speedup in replaying experiments.

5.1 Introduction

Provenance is metadata about the history of an object. Tracking the computational provenance of data has many useful applications, including ensuring the reproducibility of experiments, determining code and data dependencies when sharing research, and estimating the quality of data. Most provenance systems either require software stacks to be modified for provenance collection or require explicit modules to be installed as part of the operating systems or execution environment [32, 46, 14, 60, 54]. The efforts for adaption or modification can be a tolerable trade-off when they are applied to a single or a few machines to get provenance data. However, it is a great challenge to apply these modifications to multiple computing nodes in a network distributed experiment, and it will be a greater obstacle in heterogeneous cloud environment where different machines can require different configurations for modification. Similar to provenance acquisition, software compatibility also raises
such adaption and modification issues. Additionally, to support computation verification and optimization, such systems should allow re-execution of the software in a convenient and user-friendly manner.

In this chapter, we introduce an enhancement of the provenance auditing and packaging system PTU with features that help to capture computation provenance from an experiment running in network distributed environments. Using these new capabilities, without any system privileges, users can run their network distributed experiment within PTU and capture provenance of the experiment without any changes to the original experiment’s binaries or runtime environment. Since PTU implements network audit, users can query for network data provenance and replay those network connections offline even if the other ends of those connections were unavailable. PTU’s network-replay feature can help users in following different scenarios.

**Reduce required resources** PTU allows users to repeat a network distributed experiment by replaying processes without the need of other network-connected processes. All network communication from this process is emulated via the captured data from a referenced execution. Hence, we can re-execute all computation in a single node without the need of other computing or storage machines in the original experiment setup.

**Allow offline replay** PTU allows capture and replay an experiment that relies on inaccessible or not-always-available network related applications. These applications can be web-services that are limited to certain networks or local clusters (e.g. a local web server), privileged applications that are installed and managed by root administrators (e.g. a local Postgres database server). By capturing communication to those applications during a reference execution, PTU can replay that communication later, and as a result, can replay the whole experiment without those inaccessible applications.

This chapter makes the following contribution:
• Propose a provenance model for network data

• Design and implement a provenance capture framework for network data in distributed experiments without any modification to existing software or job manager systems

• Propose an execution replay framework to re-execute network processes offline

• Identify network-unrepeatable protocols and frameworks

The remainder of the chapter is structured as follows. We provide details of the provenance model used in PTU in Section 5.2. Section 5.3 describes the process of capturing network provenance of a network distributed experiment. We then describe our network replay implementation in Section 5.5 by describing how to use captured content of network communication combining with a numbering scheme for replaying network system calls. We evaluate our work in two experiments in Section 5.6. We conclude in Section 5.7.

5.2 Provenance Model

As the Open Provenance Model (OPM) [58] does not specify how network provenance is modeled, we introduce a new dependency between network data and processes to OPM. In recent published papers with data provenance models, new types of element, such as network vertex [54, 86], is often introduced to complement this lack of specification. In this section, we use a socket node as a specific data artifact where it stores metadata about a network TCP/IP socket (source, destination IP address and port, connected time) similar to other approaches. However, different from other proposed network provenance models, to support network replay in later re-execution, we capture actual data transferred by the socket. Hence, we define a socket node to include actual data content of a network socket (Figure 5.1). In this setup, the socket node contains both meta information and its content, similar to a data file.
In addition to the dependency between network data and processes, we also introduce a transparent dependency between remote processes and local processes. Many network distributed experiments are carried out by some workflow management systems or job management systems. On UNIX environments, these systems usually use SSH or equivalent programs to start new processes on remote machines. While the OPM has specified general process-to-process dependency, it is not entirely clear whether a remote process “wasTriggeredBy” an SSH program on the local machine, or by an SSH network connection, or by an SSH daemon on the remote machine. In our approach, we replace that unclear dependency by letting the remote Process node be linked directly to SSH with a “wasTriggeredBy” edge.

5.3 Audit Network Provenance

Using the proposed provenance model for network distributed experiment, in this section, we detail how we use our tracing mechanism to audit network provenance.

We define network communication as data transmission from one process to another using the socket interface, without involving permanent files in local or network file system or shared memory. Network communication starts by having two or more processes use sockets for data transfer. On one machine, a process listens on an agreed upon port number. Another process connects to that port number to initialize a network connection channel. After the initialization, these processes can exchange data via reading and writing to this channel. Network provenance should contain information about network communication.
these processes made during their execution.

As we aim to generate data provenance and replay information, there are two parts of network audit: **meta audit**, in which we audit for connection metadata to support provenance query, and **content audit**, in which we capture actual transferred data in network connections.

### 5.3.1 Meta Audit

In meta audit, we monitor four system calls `bind()`, `listen()`, `accept()`, and `connect()`. By default, parameters and returned results from these system calls do not contain all information about source and destination of network connections. There are different mechanisms to acquire this information. Linux command `lsof` provides list of open files (which includes socket) on a machine the command executes. However, since `lsof` returns all files and sockets from all running processes without a query interface for information of one single socket, `lsof` introduces unnecessary overhead. To overcome this issue, we use data from `/proc/net/` system directory instead (which `lsof` also uses) to obtain metadata for a Socket Node. By using `ptrace` to interrupt and stop the network system calls from exiting, we are guaranteed to have the socket alive until we finish recording the network metadata.

### 5.3.2 Content Audit

To support network replay, we capture network data transferred in each socket connection. Parameters and memory buffers from network system calls are recorded in a database as specified in Table 5.1.

To facilitate searching for recorded network system calls, network content audit implements a numbering scheme based on a *time order* of the system calls. We use this numbering scheme within each processes and within each socket number in addition with recorded time to identify corresponding connections in later re-execution. In other words, data sending
Table 5.1: Network system call audit and replay: **bold parameters** and **bold returned values** are recorded during audit and injected back for replay

<table>
<thead>
<tr>
<th>System Calls</th>
<th>Recorded/Injected Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>listen, connect</td>
<td>returned value</td>
</tr>
<tr>
<td>write, send</td>
<td></td>
</tr>
<tr>
<td>sendto, sendmsg</td>
<td></td>
</tr>
<tr>
<td>accept</td>
<td>value = int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);</td>
</tr>
<tr>
<td>read</td>
<td>value = ssize_t read(int fd, void *buf, size_t count);</td>
</tr>
<tr>
<td>recv</td>
<td>value = ssize_t recv(int sockfd, void *buf, size_t len, int flags);</td>
</tr>
<tr>
<td>recvfrom</td>
<td>value = ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);</td>
</tr>
<tr>
<td>recvmsg</td>
<td>value = ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);</td>
</tr>
</tbody>
</table>

and receiving system calls are enumerated per socket; connection initialization system calls are enumerated per processes.

5.3.3 Minimal-Setup Audit Mechanism

PTU requires minimal setup to audit experiments on a single machine or network-connected machines. To audit a computational experiment, users append PTU to the experiment command-line. Instead of executing “path-to-binary”, users execute “ptu-audit path-to-binary”. By appending PTU to the command line, PTU can attach itself to the process and audits required system calls of processes in the experiment. This audit will be continued when a process forks to create a new process as PTU also monitors `fork()` system call to audit child processes.

While it is sufficient for auditing a single machine, the `ptrace` mechanism cannot monitor a remotely spawn process. As an example, if a process uses SSH to create a new process in a remote machine, the new remote process will be started by an SSH daemon on the remote machine. This SSH daemon was usually started by process “init” that was not traced by PTU. This breaks the ability of PTU to audit the new remote process. To enable capturing information on remote machines, there are two challenges. PTU must be available on remote
machines; and programs used to start remote processes must be instrumented to include PTU tool in their execution. In particular, PTU needs to copy itself to the remote machine and get executed instead of the remote processes from the experiment. After PTU is initiated on that machine, other remote processes will be executed via PTU and be audited as described above.

Among different common process launchers, we select SCP and SSH to provide PTU copies on remote machines and implement a parser to inject PTU and its parameters. When a program wants to spawn a process on a remote machine, it invokes `execve(path-to-ssh, remote-machine, path-to-new-process, other-paramters, ...)`. PTU intercepts this `execve()` system call and performs three steps:

1. **Extract remote host parameters from execve() system call** This involves going through `execve()` parameters and looks for the first parameter that is not an SSH option or an argument of an SSH option.

2. **Copy PTU to remote host** To do this efficiently, PTU first checks if it has sent its binary to this remote host before. If not, it uses SSH to check if PTU already exists on the remote host. When both checks fail, PTU uses “scp” command to copy its binary to the remote host and stores result in a cache to avoid copying PTU to the same machine later.

3. **Inject PTU into execve() to get PTU executed before remote process** PTU modifies parameters of `execve()` to `execve(path-to-ssh, remote-machine, path-to-PTU, path-to-new-process, other-paramters, ...)` and lets the `execve()` system call to execute (Figure 5.2).

4. **Retrieve provenance records from the remote machine** Process PTU on the remote machine records provenance from remote processes of the experiment. Once the remote process finishes, those records are merged back to the machine that created those
remote processes. This operation is discussed in details in Section 5.4.

The second step of this technique is to make PTU available on remote machines. Another method to achieve this result is to use a network shared directory. Users can place PTU binary into a common shared directory among all machines, e.g. network-shared users’ home directory $HOME. Since PTU is implemented with minimal dependencies of any Linux libraries (only libc and libpthread, which are available on all recent UNIX distributions), the PTU binary is highly portable and can be run on almost any UNIX machine with recent distributions. Hence, by performing the three steps above, whenever a program tries to use SSH to execute a remote command, PTU will be copied and ready to be executed on remote machines for audit.

5.4 Distributed Provenance Database

In this section, we describe a distributed database that PTU uses to keep provenance records while auditing, and how the distributed database is merged for archive and query.

We constrain our database with several cloud-based assumption:
A1. Auditing provenance should have minimal interference with computing processes.

A2. Computing nodes and their local disks may not be available for the entire experiment duration. Users may manually or automatically add or remove computing nodes based on computing demand. Once a computing node is idle, it might be taken offline to free allocated resources or to keep computing costs low. We assume that a root node is the node that starts the experiment, and is available for permanent storage.

A3. No connectivity should be assumed among computing machines, unless there exists actual SSH connections from the experiment. A simple scenario is that a user uses machine A, to create computing jobs on machine B via SSH from A to B. Machine B is a login node of a separated cluster, and jobs on B create tasks on machine C of that cluster via SSH. It is common for machine C to be behind a firewall that prevents machines from outside of the cluster to connect to it. Hence, A cannot connect to C directly.

In order to satisfy requirement A1, provenance records are stored locally on each executing node in the auditing phase. On each executing node, once an SSH connection is established, SSH will start the experiment’s processes via a PTU process. Each of those PTU processes will create a LevelDB database for all processes that it audits. Hence, if an experiment consists of multiple tasks spawned in multiple computing nodes via SSH, each of those tasks will correspond to a database on each node. These databases and the SSH connections construct a spanning tree with the initial PTU’s database as root.

To overcome node availability in assumption A2, local databases are moved to storage that is more permanent as soon as the processes started by SSH finish. At the end of the experiment, these distributed databases should be merged back to one storage unit on the experiment’s root node. With the constraint A3, we move these databases up to their parent nodes in the database-SSH spanning tree and repeat until all local databases are merged to the database in the root node (Figure 5.3).
5.5 Replay Network Communication

Once content data in network system calls is captured, the PTU tool can replay that network communication during re-execution. We define a network replay of a computational experiment execution as a re-execution of that experiment without actual network communication but still giving the same final output. We implement network replay by replaying all network system calls executed by a process.

First, PTU locates which audited value to replace into a network system call. PTU uses the numbering scheme mentioned in Section 5.3.2 and follows three steps:

1. Match a current executing process to an audited process in database
2. Match a socket to an audited socket of that audited process
3. Match a current network system call to an audited system call of that audited socket

Once an audited system call is located, PTU replays the audited values into the current system call. Audited parameters and memory buffers are inserted into results of corresponding system call without actually invoking that call. Using ptrace, PTU intercepts 11 network
system calls, retrieves the corresponding data from its database, and returns those data into these system calls as in table 5.1. By doing so, PTU simulates socket communication for the executing process as if the process has real network sockets to exchange data.

**Network replay limitation** While our approach on replaying network communication is accurate at the socket level, computation replay-ability depends on how processes view the replayed data. We consider a process to be network-replay-able if that process makes the same network system calls in the same order, with the same passing parameters and data, expects the same returned values from those system calls and handles the returned values in the same manner. While we have test cases show success replay for HTTP data retrieval and PostGres query; in cases where the same data cannot guarantee the same process’ behaviors, replaying network communication can fail. In particular, network applications might fail to replay due to one or more of the following reasons.

1. Cryptography breaks network replay-ability of applications. Cryptography is the practice and study of techniques for secure communication in the presence of third parties. In particular, SSH [98], a public-key cryptography protocol that is used in many job management systems and data-exchange communication, cannot be replayed.

2. Data communication that uses certain types of replay-attack countermeasure: session tokens, one-time passwords, and time stamping, is not replay-able. In our observation, DNS lookup queries and answers are different for every run of the same process; hence, DNS lookup is not network replay-able.

3. Network communication that sends and receives control data instead of actual data content is not replay-able. MPICH [34] passes file descriptors between processes via UNIX socket. While PTU can replay the communication, the passing file descriptors are only valid in the reference execution and not valid in the re-execution.
Table 5.2: NASA Parallel Benchmark runtime (seconds) and overhead of PTU meta audited mode (for query) and content audited mode (for replay) compared to normal NPB execution (no PTU)

<table>
<thead>
<tr>
<th></th>
<th>Normal</th>
<th>Meta Audit</th>
<th>Content Audit</th>
</tr>
</thead>
<tbody>
<tr>
<td>NPB A.2</td>
<td>132.1±0.8</td>
<td>2.1%↑</td>
<td>5.3%↑</td>
</tr>
<tr>
<td>NPB B.2</td>
<td>530.5±0.4</td>
<td>0.8%↑</td>
<td>3.2%↑</td>
</tr>
</tbody>
</table>

5.6 Experiments and evaluation

In this section, we verify that PTU can capture distributed provenance with network communication and replay experiments that depend on unavailable online resources while keeping reasonable performance.

5.6.1 MPICH NASA Parallel Benchmark (NPB)

Figure 5.4 shows the recorded provenance of NPB [4]. The benchmark contained three applications BT-MZ, LU-MZ, and SP-MZ was compiled with parameter class A and B, for 2 processors, and run on three “m3.medium” instances of Amazon AWS 64-bit, Intel Xeon E5-2670, 1 vCPU, 3 ECU, 3.75 GiB RAM. An instance with IP address 10.164.1.135 started the MPI framework and spawned jobs to two slave instances with IP addresses 10.11.150.45 and 10.185.248.3. In this benchmark, MPICH used SSH to launch processes in other instances. In each of the two slave instances, SSH started `hydra.pmi.proxy`, which in turn started the benchmark binaries. The benchmark binaries communicated with others via network connections shown as Socket nodes in the provenance graph in Figure 5.4.

Table 5.2 shows overhead of using PTU for auditing provenance and network content. In both class A and B benchmarks, the overall overhead is not significant with meta audit introduces 1-2% increase in runtime. Content audit shows slightly higher overhead than meta audit at 3-5% increase. This difference can be explained as more network system calls were monitored in content audit; with not only metadata of network connections but also their actual transferring data was recorded.
Figure 5.4: Provenance graph with Socket node and transparent SSH process dependency
Table 5.3: PTU performance (seconds) on network related and non-network related tasks from a multi step experiment shows 3-fold reduction in re-execution time with network replay.

<table>
<thead>
<tr>
<th></th>
<th>Normal</th>
<th>Meta Audit</th>
<th>Content Audit</th>
<th>Replay</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data retrieval</strong></td>
<td>146.5±1.8</td>
<td>0.2%↑</td>
<td>134.5%↑</td>
<td>53.0±3.0 (64%↓)</td>
</tr>
<tr>
<td><strong>Other steps</strong></td>
<td>varies</td>
<td>same as PTU without network features enabled</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.6.2 *RDCEP Experiment*

In an experiment from an RDCEP paper [9], its first step is to retrieve data from external websites. PTU has successfully re-executed the experiment offline without any access to those external online resources. In Table 5.3, for the data-retrieval step, meta audit introduced almost no overhead, while content audit overhead was 134.5% due to extra time PTU spending to record actual network content data. However, replaying this experiment step showed a 64% time reduction of the original execution duration. This 3-fold speedup was due to the data retrieval step loaded network content data from local database instead of data from remote servers. Overall, depending on users’ requirements, PTU using meta audit or content audit with amortized replay tradeoff can provide reasonable performance compared to normal execution.

5.7 Conclusions

In this chapter, we have introduced new capabilities to the provenance auditing and packaging system PTU to enable network audit for provenance and replay. The tool is independent of any workflow system, works at the user level of UNIX operating systems, and thus can be adopted by most researchers. The tool is capable of capturing provenance of network data flows at different levels of granularity from as minimal as for provenance query to sophisticated capture for offline replay of network communication. Experiments show that the tool is efficient with low performance overhead and works effectively with individual applications or computational frameworks.
CHAPTER 6
CLOSING REMARKS

My dissertation work has focused on supporting authors and readers conducting reproducible research. It describes frameworks and tools, which if provided to authors can aid in assessing computational reproducibility. This chapter concludes my dissertation work with its contributions and discusses future research directions.

6.1 Conclusions

Static text-based publications summarize and market the ideas of their authors, but do not provide access to the basis for those ideas, which are often embedded in a collection of data and computational methods. Many scientific communities use the Web to share data and software packages. However, such Web resources remain largely disconnected from the claims made in papers. Readers and reviewers cannot easily reuse data and methods, assess the validity of findings, and verify results.

In this thesis, we have introduced and described SOLE, a framework for creating descriptive and interactive publications by linking them with the associated science objects. We described a suite of tools, namely CtagsCrawler, PDFExtractor, RepAudit, SOLEConnect, VMIReader, PTU, and SOLE PDF Reader. These tools can assist authors with creating science objects for source codes, datasets and online databases, workflows, and virtual machine images, while allowing readers to explore, verify, reproduce, and reuse computational experiment in research papers.

By hosting the objects on an online database, both authors and readers can interactively fetch metadata descriptions about the objects and link them with research papers. By capturing and providing software packages with provenance and an interactive PDF paper, readers can explore research papers with their associated provenance and re-execute the
corresponding experiment for the purpose of assessment, repeatability, and verification of research. We believe SOLE is a step toward a collaborative environment that engages authors and readers in conducting reproducible research.

6.2 Future Research Directions

My future work will continue focusing on tools that support authors and readers of research papers conveniently and efficiently conducting reproducible research. It covers:

- Integrating PTU’s fine-grained level of provenance with other provenance-aware systems such as database management systems and workflow management systems
- Improving the distributed provenance database and capture/replay process
- Supporting capture and replay on more operating systems

Database Provenance Integration  Sharing and reproducing experiments that involve a database is a challenge. In the past, the easiest way to share such an experiment was to manifest the database into a spreadsheet and share the resulting files of the experiment. This approach was practical when data were small. As the size of the data increase, data are often persisted in relational databases. Sharing relational databases can be tedious since it requires isolating data that is relevant for an experiment and the modifications to it.

We are looking to build a light-weight database virtualization (LDV) system, which makes users sharing experiments involving a database as simple as sharing data files with users. LDV methodically merges database provenance and operating-system provenance in PTU such that semantics from both the systems are unified without auditing redundant information or imposing additional overheads. Using PTU, creating a software package that contains relational database is completely automatic, and once shared, re-executing the experiment require no installation, configuration, or root permissions. We believe that by
merging fine-grained, DB and OS provenance, LDV can help promote reproducible research in database community.

**Workflow Provenance Integration**  Similar to database integration, integration of fine-grained operating-system provenance from PTU with existing provenance-aware workflow management systems promises better understanding of the workflow. We believe this approach can lead to re-executable papers with a better granularity level of interaction. With the PTU software packaging feature, workflow components can be decoupled from their workflow management system, which can lead to a portable workflow that can be attached to a research paper.

**Improve distributed provenance database and capture/replay process**  Our current approach to network distributed experiments are limited as stated in Section 5.5. This limitation is due to the fact that provenance information captured by PTU is at the operating system level, whereas actual data transferred between network processes can be wrapped in different network layers, such as a secured SSL layer or a MPI protocol layer. We believe that by having a *ptrace*-like interposition to appropriate libraries or interfaces, we can capture provenance without any obscurity and many limitations on network applications can be eliminated.

**Support capture and replay on more operating systems**  PTU approach is based on “ptrace” UNIX system call, which is not available on other popular operating systems such as Mac OS X and Windows. This limits the application of the PTU tool and SOLE framework. On Sun’s Solaris OS, Apples Mac OS X/Darwin, and FreeBSD operating systems, “dtrace” is a similar dynamic instrumentation tool that can support monitoring system call of processes for provenance [29, 22]. On the Windows operating system, different techniques can be used for audit and replay. DLL injection [19] can be used to install an instrument procedure to
monitor a system for certain types of events. A file system filter driver [18, 79] on Windows can intercept requests targeted at a file system. By intercepting the request before it actually reaches the file system, the filter driver can instrument the file system calls for monitoring purposes. We believed that with further investigation, these techniques can be used on those operating systems to support capture and replay feature of PTU.
REFERENCES


104


107


