Utilizing Provenance in Reusable Research Objects

Zhihao Yuan[1], Dai Hai Ton That[1], Siddhant Kothari[2], Gabriel Fils[1], and Tanu Malik[1]

[1]School of Computing, DePaul University, Chicago, Illinois, 60604
[1]Department of Computer Science, University of Chicago, Chicago, Illinois, 60637, USA
{*zhihao.yuan,dtonthat,gfils1,tmalik1*}*@depaul.edu,*
siddhant22@uchicago.edu

March 11, 2018

Abstract

Science is conducted collaboratively, often requiring the sharing of knowledge about computational experiments. When experiments include only datasets, they can be shared using Uniform Resource Identifiers (URIs) or Digital Object Identifiers (DOIs). An experiment, however, seldom includes only datasets, but more often includes software, its past execution, provenance, and associated documentation. The Research Object has recently emerged as a comprehensive and systematic method for aggregation and identification of diverse elements of computational experiments. While a necessary method, mere aggregation is not sufficient for the sharing of computational experiments. Other users must be able to easily recompute on these shared research objects. Computational provenance is often the key to enable such reuse. In this paper, we show how reusable research objects can utilize provenance to correctly repeat a previous reference execution, to construct a subset of a research object for partial reuse, and to reuse existing contents of a research object for modified reuse. We describe two methods to summarize provenance that aid in understanding the contents and past executions of a research object. The first method obtains a process-view by collapsing low-level system information, and the second method obtains a summary graph by grouping related nodes and edges with the goal to obtain a graph view similar to application workflow. Through detailed experiments, we show the efficacy and efficiency of our algorithms.

1 Introduction

Research objects—aggregations of digital artifacts such as code, data, scripts, and temporary experiment results—provide a means to share knowledge about computational experiments [1, 2]. In recent times, sharing computational experiments has become vital; scientific claims, inevitably asserted via computational experiments, remain poorly verified in text-based research papers. Research objects, together with the paper, provide an authoritative and far more complete record of a piece of research.

Several tools now exist to help authors create research objects from a variety of digital artifacts (see [3] for several tools and [4] for a variety of research objects). The tools enable research objects to be shared on websites that disseminate scholarly information, such as Figshare [5]. Despite their advantages, shared research objects do not permit easy reuse of their contents to verify their computations, or easy adaptation of their contents for reuse in new experiments. Often the extent of reuse is subject to the amount of accompanying documentation, which may be limited to compilation and installation instructions. If documentation is scant, research objects will remain unused.

The minimum use-case for sharing a computational experiment (in the form of a shared research object) involves repeating its original execution and verifying its results. To truly exploit its potential, however, it must support modified reuse. Therefore, the research object must be created and stored not as a simple aggregation of digital content, as previously advocated [2, 6], but in a readily-computable form: as a *reusable* research object. We demonstrate the distinction in two ways.

Consider a typical research paper with an analysis based on large amounts of code and data, and assume that the researcher authoring the paper has used the code and data to conduct a number of experiments that produce the paper's target figures and results. The example paper's digital artifacts relating to its experiments may be bundled together in a medium such as a file archive (.tar), compressed file format (.gz), virtual image, or container. A shared research object is free to use any of these mediums. A reusable research object, however, must use a virtual image or container, since it must produce a computational research object that, when downloaded and shared, will guarantee an instantly-executable unit of computation.

Also consider the example paper's metadata, which, similar to the metadata in most papers, is interspersed throughout the project's written analysis, and throughout its code and data. The metadata can take many forms, including annotations, version information, and provenance. A shared research object's metadata usually serves a purely informational purpose and is seldom used literally in the paper's experiments. A reusable research object, however, utilizes literal metadata by directly linking it to the code and data of the experiments. In particular, computational provenance, if collected in standard form, can guide different forms of reusable analysis—exact, partial, or modified reuse. In other words, a reusable research object can execute conditionally based on its embedded metadata, instead of simply including it as a stand-alone digital artifact that requires more interpretive labor to reason about and reuse.

Several tools to create reusable research objects have been recently proposed [7, 8, 9]. All tools use application virtualization (AV) to automatically create a container of an executable application. In AV, operating system calls during application execution are interrupted to enable the copying of all binaries, data, external user input, and software dependencies into a container. The resulting container is portable and instantly reusable: it can be run on any compatible machine without installation, configuration, or root permissions. However, the tools differ in the method of re-execution. In particular, none except Sciunit [9] captures provenance both during creation of the container and its re-execution (see Section 2 for further differences). Capturing provenance during container creation and then at each step of re-execution can be useful for a variety of purposes. In this paper, we show how provenance audited at execution and re-execution time within containers can be utilized to establish exact repetition of a previous reference execution, to construct a subset for reusing part of a research object, and to reuse contents when research objects are re-executed with different inputs.

Reusable research objects, owing to application virtualization, store an *execution trace*. To generate provenance from an execution trace, dependency information must be inferred from the trace. We show how this information can be inferred in a lazy or post hoc manner. To use inferred provenance within a reusable research object, an important consideration is the granularity at which the execution trace is audited. Auditing at the granularity of read and write of each data variables can help detect concurrent processing within application programs but imposes significant run-time overhead. Thus, we audit at the granularity of open and close of files and spawn of processes. At this granularity, given concurrent programs, exact repeatability cannot be guaranteed. For instance, if two processes read and write to the same file at the same time, in the absence of read/write provenance dependence information, the order in which they wrote to the file cannot be guaranteed. To still use provenance for container re-execution, we do not break cycles due to concurrent processing, but assume that applications are willing to re-run a few extra processes.

Even if provenance is audited at a somewhat higher granularity, it may still be too replete for comprehension and modification. In particular, when AV techniques are used to create a container, the collected provenance information, being at the file and process level, is still too fine-grained to show the overall workflow. We consider two kinds of consumers with differing objectives of using the generated provenance graph. Some expert users familiar with execution of the program would like to see the process-view of the provenance graph sans the extraneous low-level system information generated due to auditing of common libraries and system executables. Some users, alternatively, would like to see a summarized graph that is (potentially) closer in appearance to an application workflow or prospective provenance. We summarize graphs in two ways, in particular collapsing or hiding common libraries and system executables for a process-view, and summarizing retrospective provenance by finding groups of nodes and edges that are related by common ancestry to each other. Through experiments, we show that our graph summarization methods reduce the actual number of nodes and edges in graphs, by 80–91%, on average, thus producing meaningful summary graphs.

To show use of provenance in sciunits, we first describe how applications can create sciunits using the `sciunit` tool, a Python/C-based Git-like client that creates, stores, and repeats sciunits (Section 3). Our previous work [9] describes how the `sciunit` tool uses application virtualization to create containers and store multiple containers in a single sciunit using content de-duplication. In this paper, we show how to use embedded provenance for exact, partial, and modified repeatability. In particular, we focus on how provenance associated with two reference executions is matched for exact, partial, and modified repeatability, and how provenance associated with past reference executions is summarized.

The rest of the paper is organized as follows: Section 2 describes related work concerning the evolution of static and reusable research objects and how they utilize provenance. Section 3 describes the `Sciunit`—its use in applications to build containers and repeat them in various ways, and the embedded provenance model. Section 4 describes how to utilize embedded provenance for reuse—exact, partial, and modified reuse. Methods for summarizing retrospective provenance are described in Section 5. Section 6 presents our experiments. Our conclusions and future work are discussed in Section 7.

2 Related Work

In this section, we trace the evolution of research objects and how provenance is managed within different kinds of research objects.

Research objects are increasingly seen as the new social object for advancing science [10]. They are used for dissemination of scholarly work, measuring research impact, and assessing credit and attribution [11], which in the past was mostly done through research papers. The Research Object Model [2, 12] is a comprehensive standard defining the concept of a research object as a bundle of artifacts, specifying a complete digital record of a piece of research. Implementations of the standard have primarily focused on structured workflow objects [13, 14, 15], and only recently have been extended for general applications (i.e., applications executed without a formal workflow system).

To create a research object (RO) for a general application, digital artifacts must be placed within it, either manually with explicit commands or automatically by using AV. The former method is used in RO-Manager [16], a tool that uses the RO-Bundle specification [6]. A more recent approach relies on user action to create the topology, relationship, and node specifications based on a standard [17] that are eventually translated to a container [18]. In this paper, we focus on automatically creating research objects using AV.

Application virtualization is a generic approach to build research objects without modifying applications, and predominantly uses the *ptrace* or *strace* system call to create containers [19, 20]. Some prominent tools that use AV to build a research object are Sciunit [9, 21], Reprozip [7], Care [8], and Parrot [22].

Based on AV, all of these tools use *ptrace* to create a manifest of identified dependencies during application run-time. However, application re-execution differs considerably. In Sciunit [9] and Care [8], *ptrace* is also used during re-execution time to intercept system calls and redirect them within a native container. This is unlike Parrot [22] and Reprozip [7], which copy dependencies from the manifest into a *chroot* environment or Docker [23] or Vagrant [24] container for re-execution.

There are several advantages of using *ptrace* during re-execution time. A native container is available for instant reuse; if dependencies specified in the manifest have to be first copied within another container such as a Docker container, availability of a container for reuse is delayed. Experimental results show that redirecting system calls within a container leads to faster re-execution times [25]. However, more importantly, using *ptrace* enables transparent provenance auditing of the application [21] both during container creation and re-execution time. Care [8] does not audit provenance at either container creation or re-execution time. Therefore, in this paper, we have used Sciunit for understanding how to manage provenance in research objects. In this paper, we show how the audited provenance can be used for reusing research objects, in particular to verify correctness of results.

Using Sciunit does not limit the use of commercial container technologies such as Docker and Vagrant. In Sciunit, commercial containers such as Docker are merely a wrapper for standardization, since application virtualization creates a self-contained container, and the translation to Docker files from the collected dependency information is fairly straightforward. Another advantage is that Sciunit versions the contents of the containers using content de-duplication techniques [26] and thus it can produce a versioned provenance graph.

Transparent provenance auditing can be achieved at different granularities. In NoWorkflow [27], it is done at the level of the abstract syntax tree of Python programs. In Sciunit, it is independent of the programming language and at the system level. Furthermore, to support efficient containerization of application programs, Sciunit differs from PASS [28] and SPADE [29]. In particular, Sciunit audits at the granularity of file open and close and PASS and SPACE audit at the granularity of file reads and writes. Furthermore, the provenance model in Sciunit can be extended to include database applications [30] and distributed applications [31].

Understanding application execution within a research object can incentivize its re-use. Provenance audited with application virtualization methods generates fine-grained provenance, not useful for user consumption. Methods that link retrospective provenance to prospective provenance [32] are useful, except that, in most reusable research objects, there is no formal guarantee that application workflow in the form of prospective provenance shall necessarily be available. In addition, Sciunit creates containers independent of programming languages. Thus, assumptions such as users annotating source code of script files as in YesWorkflow [33] cannot be made. Therefore, we focus on methods that summarize provenance without assuming the description of prospective provenance.

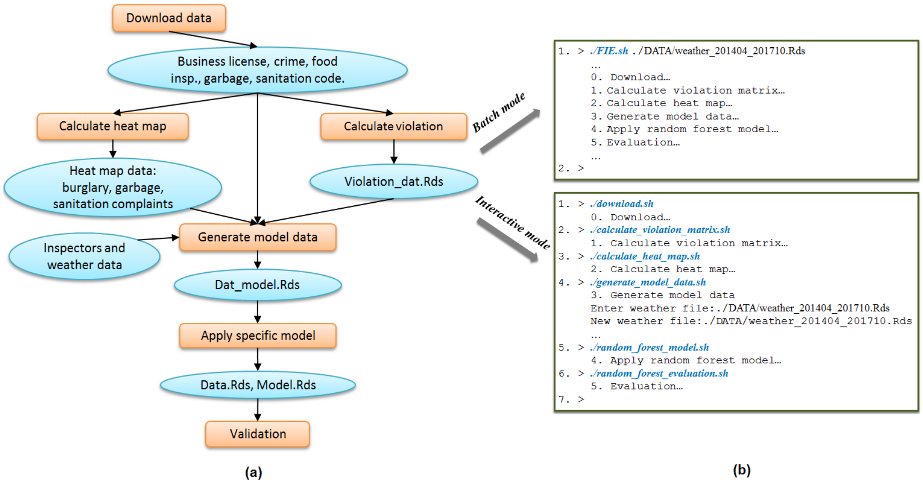
Several methods for provenance graph summarization have been proposed [34, 35, 36]. We classify them as statistical [34] and non-statistical [35, 36] methods. Statistical methods use techniques such as clustering or user-defined views to determine relevant nodes. Non-statistical methods are based on pure aggregation of nodes and derivation histories. In our experience, non-statistical methods are easier to implement and are used in light-weight databases, such as LevelDB, embedded within containers than clustering based methods, which assume presence of a graph or relational database. Therefore, in this paper, we have focused on non-statistical methods. Furthermore, within non-statistical methods, we focus on spatial summarization, i.e., reducing the number of nodes and edges in a single provenance graph and not temporal summarization i.e., summarization across multiple provenance graphs as considered in SGProv [37]. This is because the first objective is to understand application execution and therefore the objective of summarization is to generate a summary as close as possible to prospective provenance. Temporal summarization can also be useful as users compare initial application workflow with its re-runs but is currently beyond the scope of this paper.

3 Using `Sciunit`

We describe how the `Sciunit` tool is used to create reusable research objects. By design, sciunit is both the name of the reusable research object we define and the name of the command-line tool. The `Sciunit` tool creates, manages, and shares sciunits.

3.1 A Sample Application

Our reference implementation is the `sciunit`, a Python/C command-line client program that creates reusable research objects, stores them efficiently, and repeats and reproduces them [38]. To demonstrate the primary commands and salient features of the client program, we use a real-world example. Figure 1a shows an example of a predictive model used for forecasting critical violations during sanitation inspection, known as Food Inspection Evaluation (FIE) [39]. The software consists of scripts written in different languages (R and Shell) that operate on input datasets acquired from the City of Chicago Socrata data portal [40]. The output of the predictive model is continually tested using a double-blind retrodiction; the Department of Public Health conducts inspections via its normal operational procedure, which are compared with the output of the model. The pre-processing code is shared on GitHub. [41], the data is available via public repositories [40], and the predictive model analysis is also published [42]. Bundling these artifacts into a mere shared research object would likely be inefficient given data from nine different sources, which changes periodically, making analysis conducted within a certain time range obsolete. A reusable research object is needed.



```
1. > ./FIE.sh ./DATA/weather_201404_201710.Rds
     ...
     0. Download…
     1. Calculate violation matrix…
     2. Calculate heat map…
     3. Generate model data…
     4. Apply random forest model…
     5. Evaluation…
     ...
2. >
```

```
1. > ./download.sh
     0. Download…
2. > ./calculate_violation_matrix.sh
     1. Calculate violation matrix…
3. > ./calculate_heat_map.sh
     2. Calculate heat map…
4. > ./generate_model_data.sh
     3. Generate model data
     Enter weather file:./DATA/weather_201404_201710.Rds
     New weather file:./DATA/weather_201404_201710.Rds
     ...
5. > ./random_forest_model.sh
     4. Apply random forest model…
6. > ./random_forest_evaluation.sh
     5. Evaluation…
7. >
```

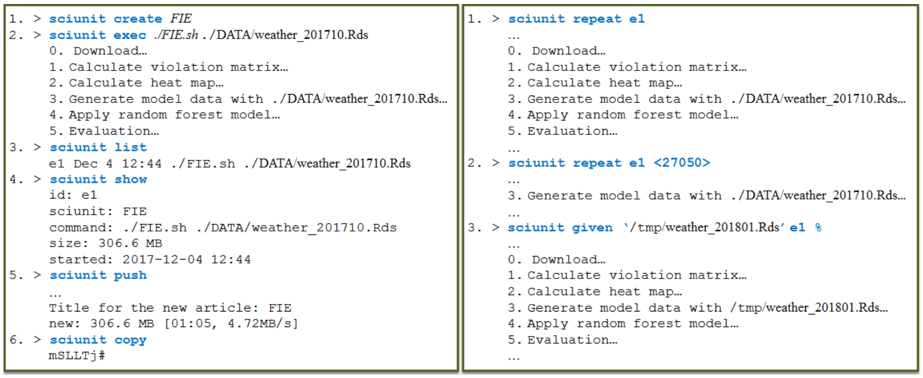
Figure 1: (**a**) conceptual view of the steps required to run the Food Inspection Evaluation [39] predictive model; (**b**) two possible execution modes.

3.2 Creating, Storing, and Repeating a Container with `Sciunit`

The FIE predictive model can be run in two modes, either as a batch mode, using a Shell script that serially executes all sub-tasks or in an interactive mode, wherein the user provides some input parameters to few sub-tasks such as weather files in a specific date range. Figure 1b shows the two possible executions. The `sciunit` client can be used to build a reusable research object consisting of identifiers of one or more re-executable containers in both the batch and interactive modes.

Figure 2a shows a sample user interaction with `sciunit` client for auditing the FIE program. The user creates a namespace sciunit titled *FIE* (Line 1). To create a container within the sciunit, the user runs the application with the *exec* command (Line 2). Packaging an application into a container also audits provenance information of the application run. Many containers, each corresponding to a given execution of the client program, can be created within the same *FIE* sciunit by using the *exec* command again. All executions can be listed with the *list* command (Line 3) and the last execution can be listed with the *show* command (Line 4).

The *exec* command makes minimal assumptions regarding the nature of the application. In particular, the user application can be written in any combination of programming languages, e.g., C, C++, Fortran, Shell, Java, R, Python, Julia, etc. or be used as part of a workflow system such as Galaxy [43], Swift [44], Kepler [45], etc. While our description assumes local execution, in practice, an application's execution can be either local or distributed. We choose an example with local execution since the AV methods for distributed and parallel applications are currently not integrated with `sciunit`, and cannot generate the required provenance graph. An AV method for database applications is outlined in Light-weight Database Virtualization (LDV) [30] and for high performance computing (HPC) programs in Pham Q. [31].



```
1. > sciunit create FIE
2. > sciunit exec ./FIE.sh ./DATA/weather_201710.Rds
     0. Download…
     1. Calculate violation matrix…
     2. Calculate heat map…
     3. Generate model data with ./DATA/weather_201710.Rds…
     4. Apply random forest model…
     5. Evaluation…
3. > sciunit list
     e1 Dec 4 12:44 ./FIE.sh ./DATA/weather_201710.Rds
4. > sciunit show
     id: e1
     sciunit: FIE
     command: ./FIE.sh ./DATA/weather_201710.Rds
     size: 306.6 MB
     started: 2017-12-04 12:44
5. > sciunit push
     ...
     Title for the new article: FIE
     new: 306.6 MB [01:05, 4.72MB/s]
6. > sciunit copy
     mSLLTj#
```

(a)

```
1. > sciunit repeat e1
     ...
     0. Download…
     1. Calculate violation matrix…
     2. Calculate heat map…
     3. Generate model data with ./DATA/weather_201710.Rds…
     4. Apply random forest model…
     5. Evaluation…
     ...
2. > sciunit repeat e1 <27050>
     ...
     3. Generate model data with ./DATA/weather_201710.Rds…
     ...
3. > sciunit given '/tmp/weather_201801.Rds' e1 %
     ...
     0. Download…
     1. Calculate violation matrix…
     2. Calculate heat map…
     3. Generate model data with /tmp/weather_201801.Rds…
     4. Apply random forest model…
     5. Evaluation…
     ...
```

(b)

Figure 2: User interaction with the Sciunit client: (**a**) audit mode and (**b**) repeat mode.

The created FIE sciunit and associated containers are stored locally unless explicitly shared with a remote repository using the *push* command, which instructs the client to upload all the containers in a sciunit to a Web-based repository (see Line 4 in Figure 2a). The `sciunit` client uses Hydroshare [46] for geoscience applications and Figshare [5] otherwise as its Web-based repository. The `sciunit` client also supports package sharing capability with *copy* command (Line 8, Figure 2a). In order to copy a package to *client2*, *client1* should have the **¡tokenID¿** generated by the command *sciunit copy* and used by *client2* to open the sciunit (i.e., *sciunit open ¡tokenID¿*). The package is transferred from *client1* to *client2* through a cloud-based web service.

A container within a sciunit (identified by an increasing sequence) can be re-run on the local machine with the *repeat* command. Users can either exactly repeat the entire computation by calling *repeat* with execution ID (see line 1, Figure 2b) or partially repeat some processes in this computation by giving a list of processes ID they want to repeat (see line 2, Figure 2b).

The option to modify data inputs or program files is also available in sciunit with the *given* command. This functionality allows users to re-execute the packages with their own local data inputs or new program files that may be stored outside container. For instance, in our example (see line 3 in Figure 2b), the *FIE* program is repeated with the new data input (i.e., "/tmp/weather_201810.Rds") at a local directory.

A sciunit may include many containers, each container corresponding to one reference execution. Each time an application is audited, duplicate file dependencies of the application can be copied into the sciunit. To avoid redundancy, Sciunit checks for duplicate dependencies as the container is created during the AV audit phase. Sciunit uses content-defined chunking to divide the container's content into small chunks identified by a hash value, as described in detail in our prior work [9].

4 Reusing Sciunits

Sciunit distinguishes between an *execution trace* and a *provenance dependency trace*. Ptracing an application generates an execution trace, which is a representation of the execution of activities in the container. However, it does not generate the correct dependency information leading to a provenance trace.

More formally, an execution trace is a labeled directed graph $G = (V, E, T)$ with nodes V and edges $E \subseteq V \times V$. Each node must be of one of the activity and entity types, wherein an activity corresponds to a process and an entity corresponds to a file. Each edge with an allowed start and end activity or entity type has a label from $\mathcal{L} = \{readFrom(file, process), hasWritten(process, file), executed(process, process)\}$, and a function $T : E \rightarrow \mathbb{T} \times \mathbb{T}$, mapping edges to intervals from a discrete time domain $\mathbb{T}$. We use $T(v_1, v_2)$ to denote the time interval associated by T to the edge (v_1, v_2) and I_b and I_e to denote the lower respective upper bound of an interval I. Thus, each edge is annotated with a time interval indicating when the two connected nodes interacted: for example, the time interval during which a process (activity) was reading from a file (entity), or a time at which a process forked another process.

An output of a application can be traced back to its input through these execution trace edges. However, connectivity in the graph does not necessarily imply dependency. Consider, the simple execution trace in Figure 3. There exists a path between A and C and the entities on that path are connected through execution edges, given the temporal constraints; however, C cannot depend on A. This is because P_2 stopped reading B before it was written by P_1. No matter what time sequence $T_1, \ldots, T_5$ is chosen. Therefore, we consider a simple inference algorithm to determine a provenance dependency trace by determining the state of a node in the G. The state of an entity e depends on an entity e' at a time T if (i) there is a path between e' and e in the execution trace; and (ii) temporal annotations on the edges of the path do not violate temporal causality. That is, there exists a sequence of times $T_1, \ldots, T_n$ so that for each path we have $T_i \leq T_i{+}1$ and $T_i \leq T(v_i, v_i{+}1)_e$. In other words, the information flows from and entity e_2 to e_1 complies with the temporal annotations.

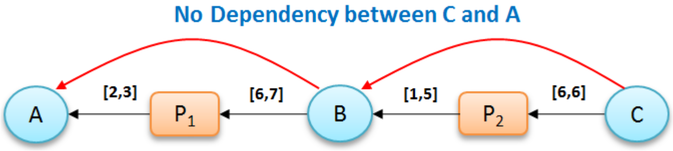


Figure 3: An example of no dependency.

Given a valid provenance dependency information, the `Sciunit` can use this to enable the various commands shown earlier. In particular, `Sciunit` can use the provenance graph to (i) simply repeat the container exactly as shared; (ii) repeat some identifiable part of the application flow; and (iii) repeat but with different input arguments, producing a different but valid output. We term them *exact*, *partial*, and *modified* repeat executions. During exact repeat execution, provenance is used for verifying if the execution was repeated exactly as the previous reference execution. During partial repeat execution, provenance is used to build a subset container containing the necessary and sufficient dependencies to run the part of the application flow. During modified repeat, provenance is used to establish which part of the container can be re-used. We describe these operations in more detail.

4.1 Exact Repeat Execution

Exact repeat execution refers to the process of running a computation again (usually on a different environment) with the same inputs and obtaining the same outputs. A container within a sciunit (identified by an increasing sequence) can be re-run exactly on the local machine with the *repeat* command (Figure 2b (Line 1)). To verify if repeat produced exactly the same outputs, the generated entities must be hashed and they must be produced in exactly the same way as they were in the reference execution.

In `Sciunit`, content validation is done through the versioning system. Even if the versioning system validates the same content, some temporary output files may have different names, and labels of processes such as its ID are not guaranteed to be identical every time application re-executed. To measure the correctness of repeatability, we focus our effort on comparing provenance graphs. Since the provenance graph records all information about the execution, having exact repeat execution means the provenance graph included in the container at audit time and new provenance graph generated during repeat execution are isomorphic.

To begin with, we first define the term provenance isomorphic as the following:

Definition 1 (**Provenance Isomorphism**)**.** Given a set of nodes $V = \{Activity(V_A), Entity(V_E)\}$ and a set of edge labels $\mathcal{L} = \{L_1, L_2, ..., L_n\}$, two provenance graphs $G = \{V, E_G\}$ and $H = \{V, E_H\}$ are said to be isomorphic if there is a non-trivial automorphism, i.e., a bijective function f : $\begin{cases} Activity(V_A) \longrightarrow Activity(V_A) \\ Entity(V_E) \longrightarrow Entity(V_E) \end{cases}$ such that $e_h = \{u, v\} \in E_H$, $Type(e_h) = L$ if and only if $e_g = \{f(u), f(v)\} \in E_G$, $Type(e_g) = L$.

In particular, there are two types of nodes: activity nodes and entity nodes according to processes and files in provenance graphs. The edge labels is based on types used in W3C PROV standard: $\mathcal{L} = \{used, wasGeneratedBy, wasInformedBy\}$.

Our Algorithm 1 describes the details of provenance isomorphism verification process. Given two input provenance graphs (i.e., G_1 and G_2), Algorithm 1 outputs a bijective function (i.e., $f : R_1 \longrightarrow R_2$) if these two graphs are isomorphic. Otherwise, it returns $False$ (Line 7). The first step of this algorithm is to calculate the $HashValues$ for each node in each graph by using function **buildHashValues** (Lines 5–6). Particularly for each node u in graph G, this function concatenates all its edge types and its neighbor labels to its $HashValues$ (Lines 9–11). Next, it turns to find a bijective function by calling **findBijection()** (Line 7). This function sequentially takes a node u_1^i in G_1 and considers each candidate u_2^i in G_2. If these two nodes both have the same type and similar $Hashvalues$ (Lines 15–17), then it recursively continues to go further with smaller graphs (Lines 18–20) until it finds a bijective function when G_1 is empty (Line 27). Otherwise, it considers other candidates in G_2 (Lines 23–25). It may also turn to $False$, if no candidate in G_2 is found (Line 26). Many other algorithms such as Nauty [47, 48], i.e., considered as the fastest general graph isomorphism algorithm, or [49] search for the whole automorphism group (all isomorphism bijections between two graphs). This is computationally hard and can lead to longer execution times. Meanwhile, our algorithm, i.e., applied for provenance isomorphism (a special kind of graph isomorphism) as defined in definition 1, may be terminated early, since having at least one bijective function is enough to claim two provenance graphs are isomorphic, and thus the repeat is considered as similar as its original execution.

The above algorithm assumes that provenance of the final output has no cycles, since repeat execution is not guaranteed if the final output has cycles. Consider a simple example in which two processes P_1 and P_2 access file F_1. P_1 runs at time t_1 and reads from F_1. After that, P_2 runs at time t_2 and writes to F_1 ($t_1 < t_2$). Since F_1 is accessed by both P_1 and P_2, it will be included in the container. However, since the content of F_1 was modified at t_2 by process P_2, process P_1 cannot be exactly repeated as the first time it was run. This problem can be avoided if the file F_1 is versioned. Suppose the container has the capability to version the resources/dependencies, and the container will keep two versions of F_1: F_1^1 and F_1^2 with respect to F_1 before and after t_2. At the repeating time, P_1 will be fed with the F_1^1 keeping the original data of F_1. Through this method, P_1 will read data exactly as the first time it was executed. Experiments show that versioning each file has an overhead. However, in `Sciunit`, it can be enabled for special cases such as when auditing a concurrent program.

4.2 Partial Repeat Execution

To partially repeat, a user selects one or multiple processes within a container. These processes are identified by their short pathname, or PID, and the user can also use the provenance graph to aid in identification. While the provenance graph can be quite detailed for a user to choose specific processes, in Section 5, we describe how a user can see a process view or a summarized application workflow akin to the workflow presented in Figure 1a from the provenance graph.

Algorithm 1: Checking the exact execution using provenance graphs

```
ProvenanceIsomorphism ($G_1$, $G_2$, $R_1$, $R_2$):
    Input  : two provenance graphs $G_1$ and $G_2$
    Output: a bijective function $f : R_1 \longrightarrow R_2$
    $R_1 = R_2 = Empty$
    if (($G_1$ is Empty) —— ($G_2$ is Empty)) then
        return False
    buildHashValues ($G_1$) /* Add hash values for each node in graph */
    buildHashValues ($G_2$)
    return findBijection ($G_1$, $G_2$, $R_1$, $R_2$) /* Find a bijection between
    two graphs */
buildHashValues (G):
    foreach node u in G do
        foreach edge $e = \{(u, v)$ or $(v, u)\}$ connects to node u do
            Add $\{Type(e), Label(v)\}$ to $u.HashValues$
findBijection ($G_1$, $G_2$, $R_1$, $R_2$):
    if ($G_1$.length # $G_2$.length) then
        return False
    foreach node $u_1^i$ in $G_1$ do
        foreach node $u_2^i$ in $G_2$ do
            if (($Type(u_1^i) == Type(u_2^i)$) && ($u_1^i.HashValues$ and
            $u_2^i.HashValues$ are similar)) then
                Remove $u_1^i$ from $G_1$ and push $u_1^i$ to $R_1$
                Remove $u_2^i$ from $G_2$ and push $u_2^i$ to $R_2$
                if findBijection ($G_1$, $G_2$, $R_1$, $R_2$) then
                    return True
                else
                    Pop $u_1^i$ from $R_1$ and add $u_1^i$ to $G_1$
                    Pop $u_2^i$ from $R_2$ and add $u_2^i$ to $G_2$
                    continue
        return False
    return True
```

Thus, for example, using the container from Figure 1, a user selects the processes "Calculate violation" and "Generate model data" as the group of processes to be partially repeated. Since this user-selected group of processes may not include all related processes needed for re-execution, we must determine these related processes, along with the data files they reference. The determined processes and files will constitute the new "partial repeat" container or "sub-container". Algorithm 2 shows the procedure for building the sub-container. It starts with the list of user-selected processes (*selectedProcs*), and progresses to include all relevant processes and files by traversing the lineage of the graph (Lines 10–24). The *getDeps* function assumes that any intermediate data files, if included as dependencies, still exist as generated from previous execution runs. The *isDecendant* function is used to detect decendant processes that need to be included. Meanwhile, the *relevantResources* function marks all data files and dependencies touched by any process in *requiredProcs*. The execution of this algorithm ensures that the data file "Heat map data" generated from the previous run of the process "Calculate heat map" is included in the sub-container, even though, in the new partial repeat execution, the process "Calculate heat map" will not be re-executed.

4.3 Modified Repeat Execution

In repeating an execution exactly, a computation is repeated with the same inputs, and obtaining the same outputs. Modified repeat execution refers to the notion of reproducibility. Reproducibility refers to the process of running a computation again with different inputs and observing the outputs. The outputs of a reproduced computation may be checked against *expected* outputs to validate the logic of the computation. Alternately, a computation may also be reproduced by altering the computational logic itself. In reproducibility, expected outputs are user-defined and, in general, hard to verify. Provenance can still be useful for modified repeat execution.

Consider the provenance graph of an execution in Figure 4. This execution consists of two processes: P and Q. Files A and C are used by P and Q, respectively. B is an 'intermediate' input produced by P and used by Q, which itself is spawned by P, and uses B and C as inputs to produce final output D. Repeating this execution would entail running it again with the exact same inputs (i.e., 'unchanged' data files) for A and C, at which the exact same result for output D will be produced. Reproducing this execution using the Sciunit *given* command implies running with a modified inputs, either A or C, or both.

If A or both inputs are modified, the entire execution must be re-run again. However, if only C is modified, then the only part of the computation that will run differently is Q (i.e., P will produce the same output B, given the same input A). If P is far more time-consuming than Q, it might suffice to run only Q, avoiding the expense of running P again. Reducing the unneeded processing time is often critical if the execution is to be altered for a large number of

Algorithm 2: Build sub-container for partial execution

```
BuildSubContainer (selectedProcs, container):
    subContainer = initialize (container)
    allProcs = getAllProcs (container)
    requiredProcs = getProcs (selectedProcs, allProcs)
    reqProcDeps = getDeps (requiredProcs)
    foreach dep in {reqProcDeps} do
        /* add dep to correct location in subContainer */
        add (dep, container, subContainer)
    return subContainer
getProcs (selectedProcs, allProcs):
    result = {selectedProcs}
    foreach proc in {allProcs} do
        foreach selProc in {selectedProcs} do
            if isDescendant (proc, selProc) then
                $result = result \cup proc$
                break
    return result
getDeps (requiredProcs):
    $result = \emptyset$
    foreach reqProc in {requiredProcs} do
        /* retrieve all related files and dependencies */
        deps = relevantResources (reqProc)
        $result = result \cup deps$
    return result
```

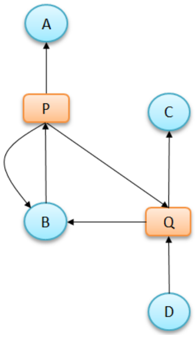


Figure 4: Provenance graph of an execution.

modifications to input C. This partial reproduction would be possible if, upon repeating the computation with its original inputs, the intermediate output B produced by P was saved in the container.

We use the embedded provenance graph to determine which part of the provenance graph need not be reprocessed again. Our algorithm is the same as Algorithm 2 in that we identify the primary processes of changed inputs, and from that determine the necessary and sufficient dependencies (i.e., *getDeps* function). This is the part of the graph that must be re-run. Nodes that are not in this dependency set are simply re-used from the container.

5 Summarizing Provenance Graphs

Provenance information generated by AV audit methods is fine-grained. A graph created from a complete set of generated provenance, using normal visualization structures such as tree or list representations, would be far too replete to be of real practical value. When viewed, this graph would present significant system-level detail that would inhibit a basic comprehension of the overall application workflow. For example, the intuitive workflow of Figure 1a, consisting of 12 nodes and 13 edges, would be represented fully as a dense graph of 146 nodes and 321 edges. Figure 5a shows a part of this replete graph redrawn for visual clarity.

The definition of 'intuitive' is subjective. We consider two kinds of consumers with differing objectives in using the generated provenance graph: (i) a predominant process-view of the provenance graph sans the extraneous low-level system information generated due to auditing of common libraries and system executables; and (ii) a summarized graph that is (potentially) closer in appearance to an application workflow or prospective provenance as in Figure 1a. In this section, we describe two formal methods for (i) and (ii). In (i), the key idea is to collapse or hide as much as possible common libraries and system executables, thus summarizing retrospective provenance; and, in (ii), the key idea is to summarize by finding groups of nodes and edges that may be related semantically to each other so as to potentially match with prospective provenance. We evaluate the first method based on total number of system information collapsed and second method based on available UML diagrams of our sample test programs. However, UML diagrams are not assumed as inputs to the summarization method.

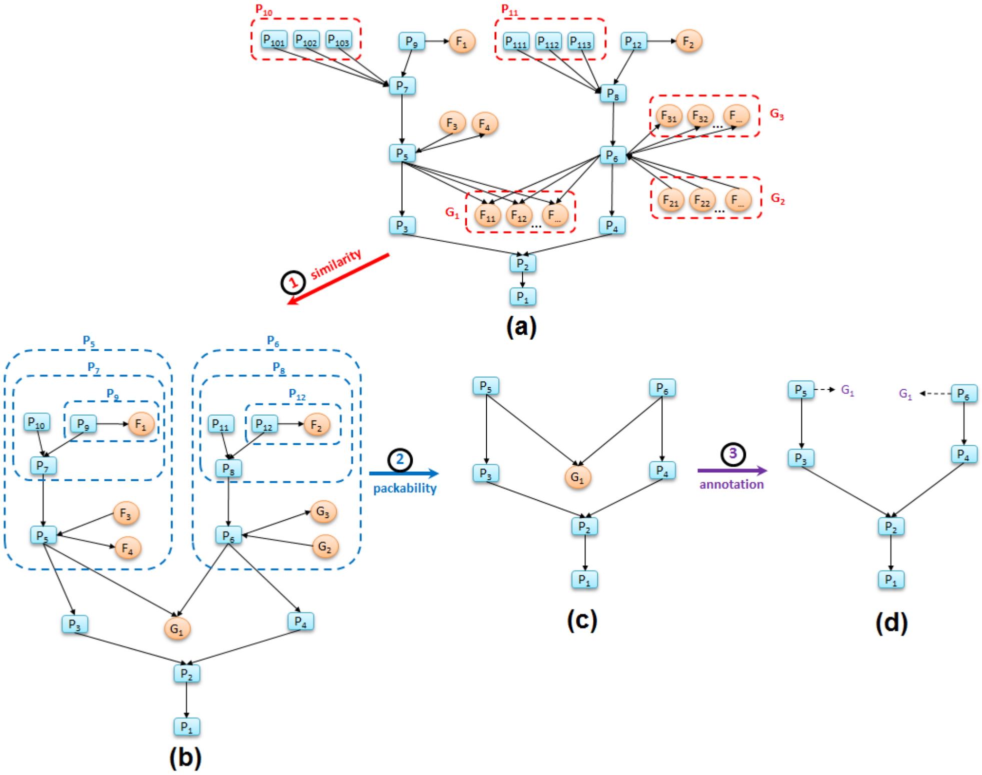


Figure 5: Graph summarization of a replete graph: (a) orginial provenance graph, (b) applying similarity rule, (c) applying packability rule and (d) final sumary graph.

5.1 Collapsing Retrospective Provenance

Given a directed graph $G = (V, E)$, where V is the set of vertices (in our graph, a vertex is of type "file" or of type "process") and E is the set of edges, we denote $Input(u)$ and $Output(u)$ as the sets of input and output edges of vertex u. Respectively, $Input(u) = \{e| \; \exists v \in V, \; e = (v, u) \in E\}$, and $Output(u) = \{e| \; \exists v \in V, \; e = (u, v) \in E\}$. The direction of an edge characterizes the dependency of its vertices. For example, a process u spawned by process v is represented by the edge (u, v), and a file u read by process v is represented by the edge (v, u). The graph G is collapsed based on the following two rules:

Rule 1. ***Similarity.*** Two vertices u and v are called *similar* if and only if they share the same type and have the same input and output connection sets: $Type(u) = Type(v)$, $input(u) = input(v)$ and $output(u) = output(v)$.

The similarity rule groups multiple vertices into a single vertex if the vertices have the same type and are connected by the same number and type of edges. Additionally, edges of similar vertices will be grouped into a single corresponding edge. Since the provenance graph follows W3C PROV-DM standard, each file is of type entity and each process is of type activity. When applied to our provenance graph, this rule groups different files and processes into the groups of files and processes.

Rule 2. ***Packability.*** A vertex u belongs to v's *generalization set* if and only if vertex u connects to v and satisfies one of the following conditions:

- Vertex u is a file that has only one edge to process v: $Type(u) = file$ and $\{\exists! e \mid e \in E \wedge (e = (u,v) \vee e = (v,u))\}$.
- Vertex u is a process that has only one output edge to process v: $Type(u) = process$ and $\{\exists! e \mid e \in E \wedge e = (u,v)\}$.
- Vertex u is a file that has only two edges—an output edge to process v and an input edge from another process x: $Type(u) = file$ and $\{\exists!(e_1, e_2) \mid (\exists x \in V,\ v \neq x) \wedge (e_1 = (u,v) \in E,\ e_2 = (x,u) \in E)\}$.

The packability rule identifies hubs in the provenance graph by packing files or processes that are connected by single edges to their parent nodes. It also packs files that are generated and consumed by a single process into their parent processes by producing a process-to-process edge.

When applied in sequence, the similarity and packability rules condense the detail-level of a graph while preserving its core workflow elements. Figure 5 (all process names and file names are simplified for brevity) illustrates how applying these two rules to a replete graph produces a graph summary that shows the primary processes in a workflow. Figure 5a presents the original replete provenance graph of one sub-task of the FIE workflow (the data processing steps "Calculate Violation" and "Calculate Heat Map" of Figure 1a). Applying the two summarization rules produces the graph in Figure 5c.

We use an annotation method that assigns higher collapsibility to file nodes than process nodes, since an application workflow is typically defined by the primary processes that it runs. Figure 5d shows how the annotation "G_1", which is a library dependency used both by "P_5" and "P_6", is attached to the two process nodes that generated it. Thus, given a file with n edges ($n \geq 2$), we replace this file with n annotations.

Collapsing system files and dependencies, may lead to some false positives. For example, Figure 6 shows the expanded view of node "P_R_27070" ("P_5" in Figure 5b). In Figure 6, similarity and packability rules group the nodes within the box into the single node "P_R_27070" (process 27070 runs a subprocess using file "21_calulate_violation_matrix.R" ("F_3" in Figure 5b) and write data to file “violation_data.Rds” (F_4 in Figure 5b)). These nodes are application nodes and not system nodes. Here, “Process_G_5” (P_7 in Figure 5b), another concealing node, correctly hides all the dependencies of the R process calculating the violation matrix.

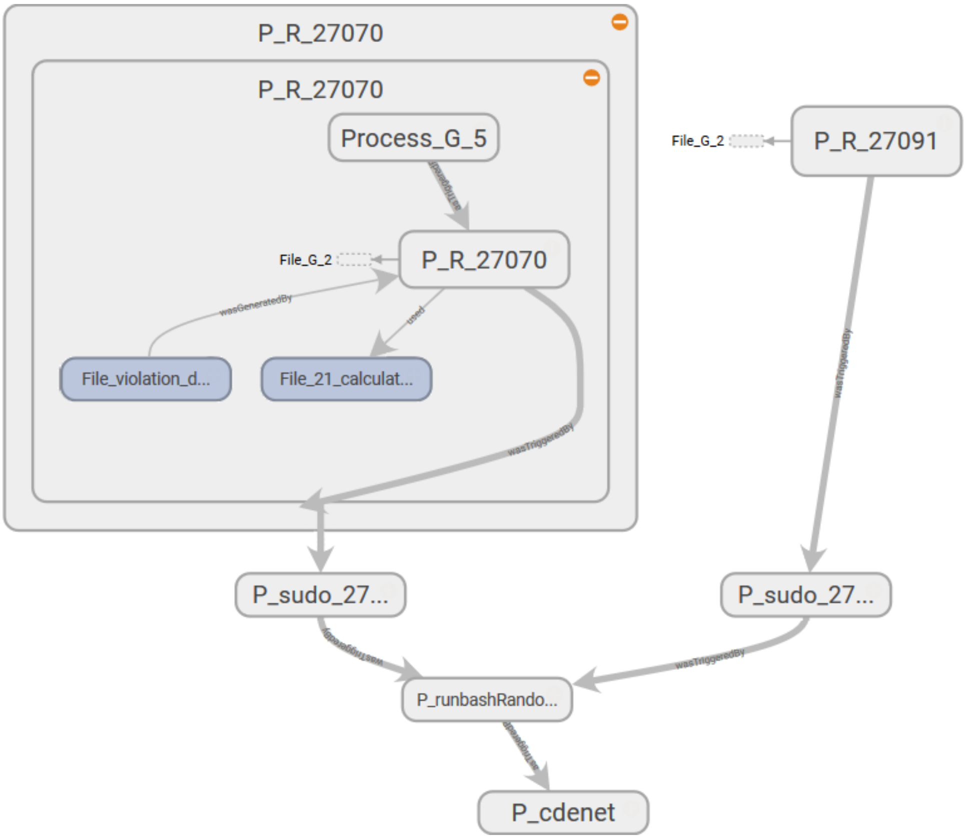


Figure 6: Expanded view of concealing node “P_R_27070” (“P_5”).

5.2 Summarizing Retrospective Provenance to Generate Prospective Provenance

The method in the previous section summarizes retrospective provenance by collapsing information. However, such summaries still differ from the conceptual view of the applications. For example, Figure 1a, which is more familiar to users, is very different in view than Figure 5d. Another equally important goal of summarization is to summarize retrospective provenance such that it (potentially) matches application workflow. In some situations, this application workflow may be available in the form of prospective provenance [32]. If available, summary methods can take advantage of this available information.

In containers created of ad hoc applications, however, application workflows are rarely available. Therefore, we describe a summarization method that determines the lineage history of nodes and uses this information to summarize retrospective provenance.

Our method to summarize retrospective provenance is based on ideas described in SNAP (Summarization by Grouping Nodes on Attributes and Pairwise Relationships) [35]. SNAP is a non-statistical method for summarizing undirected and directed graph nodes and edges based on their respective types. In brief, it first groups nodes of the same type. It then recursively sub-divides to form smaller groups of nodes that still have have the same node type but also same relationship type with other groups. SNAP considers direct relationship types amongst group nodes and not relationship types due to ancestry of the nodes. Thus, grouping provided by SNAP can be further improved for provenance graphs by considering ancestral history of nodes while grouping. If ancestral relationships are considered, then, for a node, ancestors or descendants will not be grouped together since, by definition, the ancestral history of an ancestor and its descendent is different. Similarly, nodes in the same group will not share any relationship because then their ancestral history will be different.

To identify nodes with the same derivation history, first nodes of the same type are grouped, defined as

Definition 2 (Node Grouping). Given a provenance graph $G(V, E)$, $\phi = \{G_1, G_2, \ldots, G_k\}$ is a node-grouping such that

(1) $\forall G_i \in \phi, G_i \subseteq \mathit{Activity}(G)$ or $G_i \subseteq \mathit{Entity}(G)$, and $G_i \neq \emptyset$,

(2) $\cup_{G_i \in \phi} G_i = V(G)$,

(3) $\forall G_i, G_j \in \phi$ and $(i \neq j), G_i \cap G_j = \emptyset$.

In particular, in (1), node grouping is over nodes of two types: activity nodes and entity nodes, in (2), the union of all node grouping is equal to the nodes in G; and, in (3), given a node grouping, groups are not overlapping, but distinct.

We now consider grouping G by ancestry. For this, we identify the ancestors of a node as follows. For a given grouping ϕ, the ancestors of a node v is the set $\mathit{Ancestor}_{\phi,E}(v) = \{(\mathit{Ancestor}(\phi(u)), \mathit{Type}(u, v)), (u, v) \in E, \mathit{Type}(u, v) \in \mathcal{L}\}$. Type of edges is based on types used in W3C PROV standard where $\mathcal{L} = \{used, wasGeneratedBy, wasInformedBy\}$. Nodes that do not have an ancestor are assigned the start node as an ancestor, with a start label edge. Now, we define grouping nodes by ancestry.

Definition 3 (ancestry grouping). A grouping $\phi = \{G_1, G_2, \ldots, G_k\}$ has the same ancestry if it satisfies the following:

(i) Node Grouping Definition 2,

(ii) $\forall u, v \in V(G)$, if $\phi(u) = \phi(v)$, then $\forall L_i \in \mathcal{L}$, $\mathit{Ancestor}_{\phi,L_i}(u) = \mathit{Ancestor}_{\phi,L_i}(v)$.

Figure 7a shows the grouping of nodes due to SNAP, in which nodes with the same types and same ancestors are separated into different groups if their descendants are different, and due to ancestry grouping in Figure 7b, in which nodes of the same type with same ancestors remain grouped.

Ancestry grouping, however, may group system files and dependency information together with application-specific nodes. Consider the example in Figure 8. Suppose that we have a provenance graph on the left of Figure 8 that has three nodes of process (P_1, P_2 and P_3), four nodes of file (F_1, F_2, F_3 and F_4) and six edges (relationship *used*). The summary ancestry graph shown in center of Figure 8 dividing the original graph into two groups $g_1 = \{P_1, P_2, P_3\}$ and $g_2 = \{F_1, F_2, F_3, F_4\}$, satisfies 3. (All nodes in every group have similar node types and associate with the same ancestry groups). However, from the conceptual point of view, file F_4, a dependency used by other processes is different from other file nodes in g_2, since this node is the only node that associates with all processes in group g_1. In other words, F_4 has $InDegree_{g_1,used}(F_4) = 3$ while other nodes in g_2 have $InDegree_{g_1,used}(F_1) = InDegree_{g_1,used}(F_2) = InDegree_{g_1,used}(F_3) = 1$.

To uniquely differentiate P_4, we define ancestry-degree compatible grouping to summarize provenance graphs.

Definition 4 (ancestry-degree grouping). A grouping $\phi = \{G_1, G_2, \ldots, G_k\}$ has the same ancestry-degree if it satisfies the following:

- *Ancestry grouping Definition 3,*
- $\forall u, v \in V(G)$, *if* $\phi(u) = \phi(v)$, *then* $\forall L_i \in \mathcal{L}$, $InDegree_{\phi,L_i}(u) = InDegree_{\phi,L_i}(v)$ *and* $OutDegree_{\phi,L_i}(u) = OutDegree_{\phi,L_i}(v)$, *where* $InDegree_{\phi,L_i}(u) = |\{v|v \in \phi, (v,u) \in E, Type(v,u) = L_i\}|$ *and* $OutDegree_{\phi,L_i}(u) = |\{v|v \in \phi, (u,v) \in E, Type(u,v) = L_i\}|$.

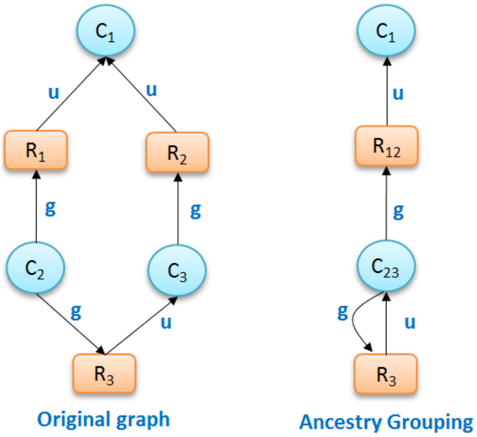


Figure 7: An example of ancestry grouping.

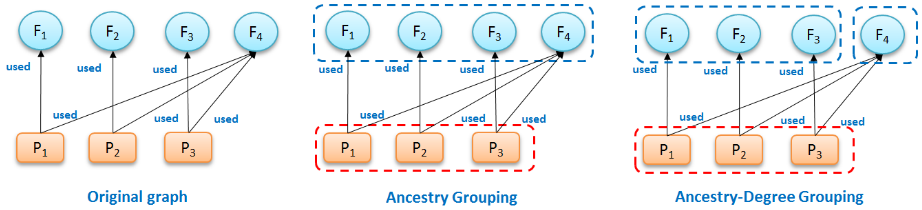


Figure 8: An example of ancestry-degree compatible grouping.

Based on this definition, the summary graph shown in Figure 8 (right) composed of three groups—$g_1 = \{P_1, P_2, P_3\}$, $g_2 = \{F_1, F_2, F_3\}$ and $g_3 = \{F_4\}$—is ancestry-degree compatible because it is Ancestry-Grouping compatible and all the nodes in every group have the same number of output edges from/to other groups (all nodes in g_2 have one input edge from group g_1, all nodes in group g_1 have two output edges: one to g_2 and one to g_3).

We now describe the summary algorithm, which, given a retrospective provenance graph, produces an ancestry-degree grouping. In this algorithm, we first divide the nodes into group with same node type $\{g_i\}$ and store them in a stack Φ (Line 2). Next, for each group g in the stack, we re-partition all groups in $\{g_i\}$ in stack by calling function **divideGroup()** with a list of vertices in g (i.e., Lines 5–7). Function **divideGroup()** is called twice with different direction parameters, since it is applied on two different directions of edges (i.e., input or output). The main purpose of this function is to re-organize all the groups (i.e., $\Phi_c = \Phi - \{g\}$) that are relevant to g by checking all the vertices and edges of vertices in group g of stack Φ (Lines 12 and 26). First, for each types of edges or relationships (in our context, there are three types of edge: $\{used, wasGeneratedBy, wasInformedBy\}$), it calculates the number of edges (or "degree") from/to vertices in each group g_i of Φ to/from a vertex v in V_g (Lines 16–22). Second, it further divides these vertices u_i in each group g_i of Φ_c by considering the degree of these vertices (i.e., the number of edges from/to vertices in group g). This means vertices that belong to the same group must have the same degree (Line 22). Third, we add all the new generated groups $\{g\_par_i\}$ to Φ if they have never been in Φ (Lines 24–26), before the next consideration of other groups in Φ. Finally, once all the groups g in Φ are considered, we obtain the summary graph Φ_S by joining all the groups together (Lines 8–10).

Figure 9 presents an example of graph summarization. The left figure (i.e., Figure 9a) shows the work-flow of FIE [39] application drawn by users that describes the conceptual view of FIE application, while the right figure (i.e., Figure 9b) shows the provenance summary graph of FIE application after applying ancestry-degree grouping. As a general observation, these two graphs are fairly close to each other. The summary graph almost captures all the information about the application that users might need at the general view. There are

Algorithm 3: Ancestry-degree grouping

```
Ancestry-degree grouping (Labeled G (V, E)):
    Group all {nodes} with same attributes into $\{g_i\}$ and store all groups in a Stack $\Phi = \{g_i\}$;
    while ($\Phi$ is not empty) do
        $g = \Phi$.pop ();
        $V_g$ = vertex list of $g$;
        divideGroups ($\Phi$, $V_g$, "from");
        divideGroups ($\Phi$, $V_g$, "to");
    foreach node in {nodes} do
        /*groupofNode() returns the group where given node belongs to*/;
        $\Phi_S = \Phi_S \bigcup$ groupofNode (node);
    return $\Phi_S$;
divideGroups ($\Phi$, $V_g$, direction):
    foreach types of edges or labels do
        /*$g_i = \{u_i\}$ has at least one vertex $u$ connecting to at least one vertex $v$ in $V_g$*/;
        foreach group $g_i$ in $\Phi$ and $g_i$ connects to $V_g$ do
            foreach vertex $u$ in group $g_i$ do
                Let $e$ is an edge that connects between vertex $u$ and a vertex $v$ in $V_g$;
                if (direction == "from") then
                    $e = (u, v)$;
                else
                    $e = (v, u)$;
                Count the number of edges $e$ (i.e., "degree" of $u$);
            Re-partition all vertices $\{u_i\}$ in group $g_i$ into a list of sub-groups {g_par} with the same degree;
            foreach group $g\_par_i$ in {g_par} do
                if ($g\_par_i$ not in $\Phi$) then
                    $\Phi$.push ($g\_par_i$);
```

some minor differences between them. For example, the two processes "Calculate heat map" and "Calculate violation" are clearly separate in the left figure while in the right figure they are grouped together. Similarly, the two groups of files "heat map data" and "Violation_dat.Rds" are separate in the left figure and grouped in the right. These groups of files and groups of processes are ancestry-degree compatible, and thus they are grouped in the right figure. While they are separate in the application workflow, ancestry-grouping is helpful in general. For instance, the ancestry-grouping grouped all data files into a single group at the top.

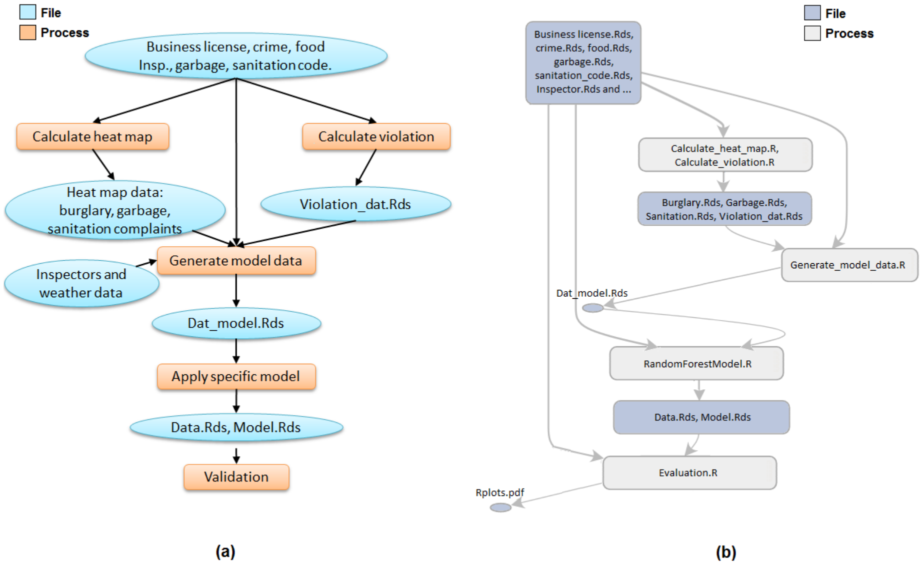

File
Process
Business license, crime, food Insp., garbage, sanitation code.
Calculate heat map
Calculate violation
Heat map data: burglary, garbage, sanitation complaints
Violation_dat.Rds
Inspectors and weather data
Generate model data
Dat_model.Rds
Apply specific model
Data.Rds, Model.Rds
Validation
(a)
File
Process
Business license.Rds, crime.Rds, food.Rds, garbage.Rds, sanitation_code.Rds, Inspector.Rds and ...
Calculate_heat_map.R, Calculate_violation.R
Burglary.Rds, Garbage.Rds, Sanitation.Rds, Violation_dat.Rds
Generate_model_data.R
Dat_model.Rds
RandomForestModel.R
Data.Rds, Model.Rds
Evaluation.R
Rplots.pdf
(b)

Figure 9: An example of using ancestry-degree grouping. (a) Original user work-flow of FIE and (b) Provenance summarization graph

6 Experiments

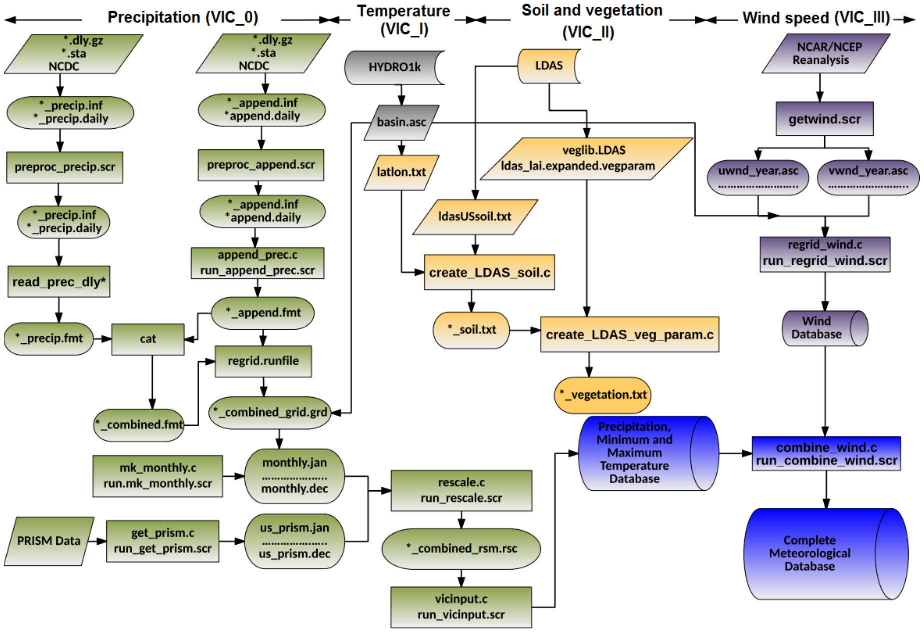
The true usefulness of sciunits can only be measured by their adoption. Efficiency of creating sciunits can be a driving force in adopting the use of sciunits over traditional shared research objects. When an efficiently-versioned, easily-created sciunit is shared, along with an embedded, self-describing application workflow, we believe the probability for reuse will greatly increase. In this section, through two complex real-world workflows, we quantify the performance of packaging and repeating sciunits, the time and space overheads of storing them, and the efficiency of reusing them utilizing integrated provenance visualizations. We implemented our `sciunit` client in Python and C. The source code and documentation of `sciunit` is available from [38].

6.1 Use Cases

We consider two real-world use cases for experimental evaluation: (i) the Food Inspection Evaluation (FIE) [39] workflow, a computationally-intense use case that has been the running example in our paper, and (ii) the Variable Infiltration Capacity (VIC) [50] model, an I/O-intensive (Input/Output-intensive) data preprocessing pipeline for a hydrology model taken from `geotrusthub.org`.

The first use case is notable for its transparency in its rigorous inspection audits, owing to the influence of the Open Data movement within the City of Chicago. The second use case is a highly-relevant test bed for sciunits: the VIC model is very popular in the hydrology community, and its data preprocessing pipeline, which relies heavily on legacy code, is notoriously difficult to reassemble [50].

Tables 1 and 2 describe the details of FIE and VIC in terms of source code file programming languages, number of source code and data files, number of program files required as dependencies, and total application sizes (both FIE and VIC have four sub-tasks, labeled 0, I, II, and III, which are described below). Figures 1a and 10 show conceptual views of the application workflows for the two use cases.



Precipitation (VIC_0)
Temperature (VIC_I)
Soil and vegetation (VIC_II)
Wind speed (VIC_III)
*.dly.gz *.sta NCDC
*_precip.inf *_precip.daily
preproc_precip.scr
*_precip.inf *_precip.daily
read_prec_dly*
*_precip.fmt
cat
*_combined.fmt
*.dly.gz *.sta NCDC
*_append.inf *append.daily
preproc_append.scr
*_append.inf *append.daily
append_prec.c run_append_prec.scr
*_append.fmt
regrid.runfile
*_combined_grid.grd
mk_monthly.c run.mk_monthly.scr
monthly.jan .... monthly.dec
PRISM Data
get_prism.c run_get_prism.scr
us_prism.jan .... us_prism.dec
HYDRO1k
basin.asc
latlon.txt
rescale.c run_rescale.scr
*_combined_rsm.rsc
vicinput.c run_vicinput.scr
LDAS
veglib.LDAS ldas_lai.expanded.vegparam
ldasUSsoil.txt
create_LDAS_soil.c
*_soil.txt
create_LDAS_veg_param.c
*_vegetation.txt
Precipitation, Minimum and Maximum Temperature Database
NCAR/NCEP Reanalysis
getwind.scr
uwnd_year.asc ....
vwnd_year.asc ....
regrid_wind.c run_regrid_wind.scr
Wind Database
combine_wind.c run_combine_wind.scr
Complete Meteorological Database

Figure 10: Conceptual view of the VIC workflow [50] ((*) in this figure denotes a list of files that share the same prefix or suffix).

Each use case is characterized by a shareable model, in which each step is conducted independently by one user, and subsequently shared with another user who builds upon or forks the shared workflow in the following step. Thus,

Table 1: Food inspection evaluation sub-task applications.

	FIE_0	**FIE_I**	**FIE_II**	**FIE_III**
Source code languages	R, Bash	R, Bash	R, Bash	R, Bash
Source code files	19	20	24	29
Data files	2	8	14	14
Dependency files	255	255	411	659
Size of all files	133.2 MB	178.4 MB	289.7 MB	306.6 MB
Normal run time	52.046 s	238.833 s	295.785 s	7200 s

Table 2: Variable infiltration capacity sub-task applications.

	VIC_0	**VIC_I**	**VIC_II**	**VIC_III**
Source code languages	C, C++, Python, C shell, Fortran			
Source code files	35	61	77	97
Data files	3689	6313	11,460	11,481
Dependency files	247	260	314	357
Size of all files	1.2 GB	1.3 GB	2.2 GB	2.3 GB
Normal run time	158.734 s	306.069 s	363.147 s	377.29 s

the FIE workflow, for example, is broken down into the following sub-tasks, each encapsulated in a single application: (i) FIE_0, which calculates a heat map from downloaded inspection records; (ii) FIE_I, which processes the heat map to generate data model inputs; (iii) FIE_II, which applies a specific model and validates it; and (iv) FIE_III, which downloads the original inspection records and applies an end-to-end validation routine to the previous three sub-tasks. The download process of subtask iv is often the most time-consuming step.

The main `sciunit` client was implemented in Python and C. Sciunit's versioning tool was written in C++, using the block-based deduplication techniques proposed in [26] and [51]. `sciunit`'s provenance graph visualization was written in Python, using libraries from TensorBoard [52]. All sciunit client *exec* and *repeat* experiments, along with their baseline normal application runs, were conducted on a laptop with an Intel Core i7-4750HQ 2.0 GHz CPU, 16 GB of main memory, and a 1 TB SATA SSD (Solid state disk), running the Arch Linux 64-bit OS (at Chicago, Illinois, 60604, USA).

6.2 Creating Sciunits

Tables 1 and 2 present the baseline normal execution times for the sub-tasks of the two use cases. We note that each application encompasses substantial resources (in the form of code and data), has many external dependencies, and is also characterized by lengthy CPU-and-memory-intensive tasks. Additionally, the nature of FIE's processing tasks differ significantly from those of VIC. FIE front-loads its input data sets into memory, and then utilizes machine-learning logic to process its data. VIC also runs many intricate calculations, but differs from FIE in that it interlaces file input and output operations regularly throughout its code. This difference will be key in understanding that sciunits have minimal performance impact on most—but not all—types of applications.

Figure 11 compares the baseline normal execution time of each subtask (Test results for the FIE_III and VIC_III sub-tasks were omitted due to significant amounts of network-dependent downloading operations.) with the time consumed by packaging the sub-task with the `sciunit`'s *exec* command, and with the time consumed by repeating the sub-task with the `sciunit`'s *repeat* command. We note that the performance impact of auditing and repeating on FIE's run times was negligible: auditing FIE with *exec* resulted in only a 3.6% time increase, and executing FIE with *repeat* added only a 1.3% increase to run time. The overhead cost of auditing and re-executing a package are in the monitoring of the I/O operations. Meanwhile, in FIE, the I/O access time is much less than CPU processing time. Also note that our tests were done on SSD offering much better performance than HDD (hard disk drive). The reasons explain why the overheads in these cases are negligible. Conversely, both packaging and repeating VIC with `sciunit` each nearly doubled the original application run times: as noted in the preceding paragraph, it was evident that using `sciunit` with I/O-intensive (Input/Output-intensive) applications affected application performance significantly. In addition, as mentioned in Section 4, the overhead cost of versioning files and processes in the container significantly impact the application execution time (compared to the execution time shown in Figure 11). However, this cost is reasonable when compared to the size of the VIC application.

We obtain one further observation from these experiments by comparing each application *exec* time with its corresponding *repeat* time. Compared to application repeat increases, auditing increases were slightly higher. This difference can be understood by examining sciunit's behavior during AV audit-time: auditing entails copying an application's code and data into a sciunit container, but running the sciunit container with *repeat*, however, only redirects to these copied files, and therefore precludes the file copy time.

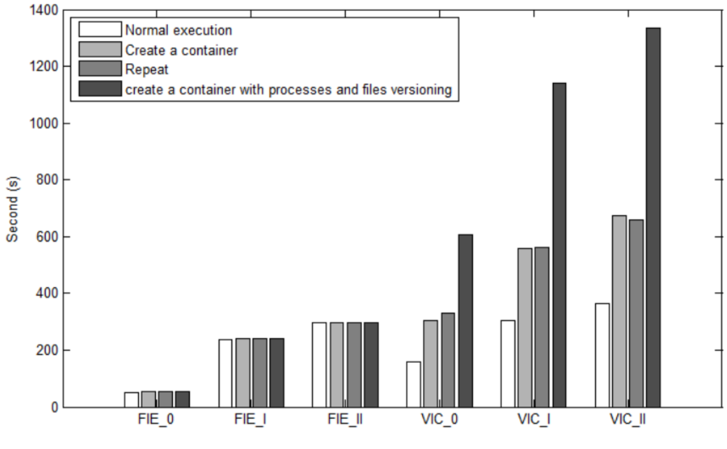


Figure 11: Execution times for normal runs, creating containers, and repeating.

Table 3: Exact repeatability evaluation.

	Repeatability		**Execution Time (s) (Measured on Our above Described Machine)**	
Selected Applications	**Re-execution (Tested on Ubuntu, Arch Linux, Hydroshare Jupyter Notebook, Amazon EC2, etc.)**	**Provenance Graph Isomorphism**	**Execution**	**Repeat**
FIE_0	Succeed	Matched	52.046	53.954
FIE_I	Succeed	Matched	238.833	240.014
FIE_II	Succeed	Matched	295.785	297.681
FIE_III	Succeed	Matched	7200	7308.6
VIC_0	Succeed	Matched	158.734	606.21
VIC_I	Succeed	Matched	306.069	1140.5
VIC_II	Succeed	Matched	363.147	1332.98
VIC_III	Succeed	Matched	377.29	1384.89

6.3 Repeatability Evaluation

To measure the exact repeat execution, we run all test cases presented in Tables 1 and 2 on many different environments such as Ubuntu, Arc Linux, Hydroshare Jupyter Notebook, Amazon EC2, etc. The results are shown in Table 3.

As clearly shown in Table 3, these applications can be repeated successfully in all tested environments. We also applied Algorithm 1 to verify the provenance graph isomorphism between original runs and re-executions. The results are recorded in the column "Provenance graph isomorphism". Since our isomorphism algorithm will finish when the first bijection is found, its performance is good even for large provenance graphs (e.g., it takes less than one second for handling with provenance graphs having up to 150 nodes and 320 edges).

6.4 Partial and Modified Repeat Execution

The main ideas of partial reproducibility are to reduce both execution time (only execute the necessary parts) and container size (not to include the data files or dependencies that will not be used). Therefore, we measure the partial reproducibility by the following criteria: (i) correctness; (ii) resource usability and (iii) execution time.

Table 4 shows our evaluation on partial and modified reproducibility on the two selected use cases (i.e., FIE_III and VIC_III). In particular, we built the partial and modified containers on the originals of FIE_III and VIC_III. In FIE_III, we selected only the process ID that calculates the heat map from the downloaded file, and then built the sub-container (i.e., FIE_Par) using Algorithm 2. Meanwhile, the modified execution of FIE_III (i.e., FIE_Mod) was tested when we change the inputs (i.e., use new weather data file: "/tmp/weather_201801.Rds") using the *given* command (see Section 3). Similarly, in VIC_III, we built the partial container (VIC_Par) with the process ID that only processes precipitation data. Table 4 shows the number of files and dependencies within the partial containers in comparison with those of original containers, as well as the

Table 4: Partial and modified repeat executions.

	Original, Modified and Partial Executions of FIE_III			Original, Modified and Partial Executions of VIC_III		
Information	**FIE_III**	**FIE_Mod**	**FIE_Par**	**VIC_III**	**VIC_Mod**	**VIC_Par**
# of source files	29	29	19	97	97	35
# of data files	14	14	2	11,481	11,481	36
# of dependencies	659	659	255	357	357	247
Executable	Succeed	Succeed	Succeed	Succeed	Succeed	Succeed
Execution time (s)	7200	-	52	377	-	159
# of files not used	0	-	0	0	-	0

differences in runtimes of repeat and original run. Values from row #of files not used denote that all files in the partial containers are touched when the application runs, indicating no redundancy in these partial containers. Meanwhile, row Executable shows the successfulness of partial and modified repeatability.

6.5 Reusing Sciunits with Provenance Visualizations

Application virtualization has traditionally led to fine-grained provenance graphs that are often difficult to decipher. In this sub-section, we determine if our summarization rules produce a usable provenance graph that is closer to a theoretical, intuitive user application workflow. We focus this discussion on experiments for the FIE sub-tasks, but note that experiment results for the VIC sub-tasks were similar.

To evaluate the effectiveness of summarization, we first considered three traditional, replete (i.e., fine-grained) provenance graphs generated by the legacy Provenance-To-Use (PTU) [21] application on auditing FIE_I, FIE_II, FIE_III (We did not consider FIE_0 in this analysis since its original replete graph was too small and simple to benefit measurably from summarization.). We calculated the number of nodes (each a process or a file) and edges present in each replete graph. Next, we calculated the number of nodes present in the corresponding `sciunit` container provenance graphs (these latter, summarized graphs were produced by using both the similarity and packability rules (i.e., collapsing retrospective provenance method) and ancestry-degree grouping method). Figure 12 depicts a comparison of these methods (i.e., original, collapsing retrospective provenance method and ancestry-degree grouping method). It should be noted that the same provenance logs (produced by PTU on audit) were used in the generation of these sets of graphs.

Graph summarization reduced the number of file nodes, process nodes, and edges by averages of 88%, 41%, and 87% with graphs generated by collapsing retrospective provenance method and 90%, 91% and 93% with graphs generated by ancestry-degree grouping method. On closer examination, since the anno-

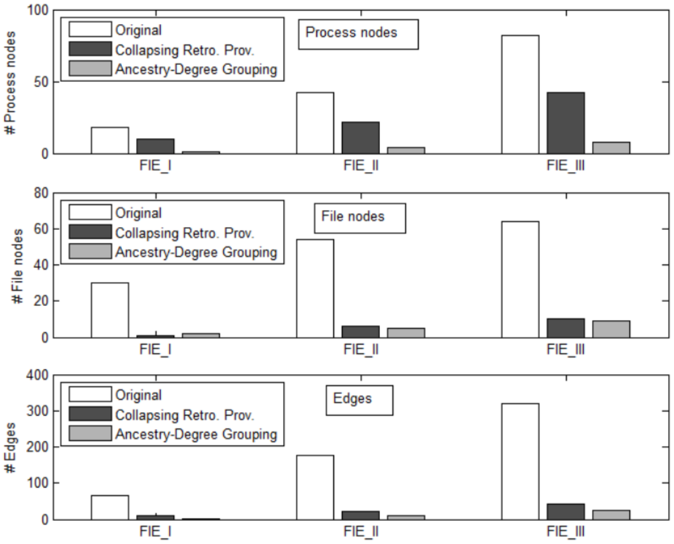


Figure 12: Number of nodes and edges in original and summarized graphs.

tation technique only applies to files and their associated edges, we observed a larger decrease in the number of files and edges than the decrease in the number of processes. Of crucial—but less measurable—importance, we noted that the much smaller number of nodes and edges of the summarized graphs also carried more meaningful, intuitive labels, similar to those in Figure 5.

We also measured the number of clicks needed to expand summarized graphs to replete graphs. For FIE_III, which had the largest graph, expanding any summarized node required a maximum of four user clicks to reach its replete view. Expanding all the nodes in this large graph took 45 clicks. This observation showed that graphs were summarized very well spatially and intuitively, yet still capable of allowing fully-detailed provenance examination with a modest amount of user interaction.

As seen in Figure 12, there are some differences between summary graphs from the collapsing retrospective provenance method and ancestry-degree grouping method. In general, applying the ancestry-degree grouping method is more efficient than collapsing retrospective provenance method in terms of number of objects. However, the graphs from collapsing retrospective provenance method are still clear and it is easy to understand the system detailed information. Indeed, the key differences between these two are the messages they deliver (see summary graphs in Figures 5 and 9). The summary graph from collapsing retrospective provenance method describes how an application be executed with its dependencies. Meanwhile, the one from ancestry-degree grouping method illustrates the conceptual view of an application. Therefore, we extend the new summarization method while keeping the old version and let users select between these methods according to which information they would prefer to examine.

7 Conclusions

Computational reproducibility [53] is a formidable goal requiring advancements in policy [54], user perception [55], and reproducible practices and tools [3]. As we embrace this goal within the sciences [56], we have encountered that computational provenance is the key to enhancing the experience of reproducible packages as created by the use of application virtualization. In this paper, we have outlined methods to create and store containers based on application virtualization and demonstrated an easy-to-use Git-like client, the `sciunit` that enables reproducibility for a wide variety of use cases. We showed how embedded provenance can be used to reuse the sciunit and understand them by summarizing embedded provenance. The field of computational reproducibility is a moving target and there are emerging requirements to use provenance to address reproducibility within Jupyter notebooks [57] , Matlab, distributed data-intensive programs, and parallel HPC applications, which we hope to address as part of future work.

References

[1] T. Miksa, “Using ontologies for verification and validation of workflow-based experiments,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 43, no. 1, 2017. [Online]. Available: http://www.websemanticsjournal.org/index.php/ps/article/view/488

[2] K. Belhajjame, J. Zhao, D. Garijo *et al.*, “Using a suite of ontologies for preserving workflow-centric research objects,” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 32, pp. 16–42, 2015.

[3] V. Stodden, F. Leisch, and R. D. Peng, Eds., *Implementing Reproducible Research.* CRC Press, 2014.

[4] T. Malik, Q. Pham, and I. T. Foster, “SOLE: Towards Descriptive and Interactive Publications,” *Implementing Reproducible Research*, vol. 33, 2014.

[5] Figshare.com, “Figshare,” https://figshare.com/, 2017, [Online; accessed 2-May-2017].

[6] S. Soiland-Reyes, M. Gamble, and R. Haines, “Research object bundle 1.0,” https://researchobject.github.io/specifications/bundle/, 2014, [Online; accessed 2-May-2017].

[7] F. Chirigati, D. Shasha, and J. Freire, “Reprozip: Using provenance to support computational reproducibility,” in *Proceedings of the 5th USENIX Conference on Theory and Practice of Provenance*, ser. TaPP’13. Berkeley, CA, USA: USENIX Association, 2013, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2482613.2482614

[8] Y. Janin, C. Vincent, and R. Duraffort, “Care, the comprehensive archiver for reproducible execution,” in *Proceedings of the 1st ACM SIGPLAN Workshop on Reproducible Research Methodologies and New Publication Models in Computer Engineering*, ser. TRUST. New York, NY, USA: ACM, 2014, pp. 1:1–1:7.

[9] D. H. Ton That, G. Fils, Z. Yuan, and T. Malik, “Sciunits: Reusable research objects,” in *IEEE eScience*, Auckland, New Zealand, 2017.

[10] D. De Roure, “Towards Computational Research Objects,” in *ACM Workshop on Digital Preservation of Research Methods and Artefacts*, 2013.

[11] Yale Roundtable, “Reproducible Research,” vol. 12, pp. 8–13, 2010.

[12] S. Bechhofer, I. Buchan, D. De Roure, P. Missier *et al.*, “Why linked data is not enough for scientists,” *Future Generation Computer Systems*, vol. 29, 2013.

[13] O. Corcho, D. Garijo Verdejo, K. Belhajjame *et al.*, “Workflow-centric research objects: First class citizens in scholarly discourse.” 2012.

[14] D. De Roure, K. Belhajjame, P. Missier, J. Gmez-Prez *et al.*, “Towards the preservation of scientific workflows,” in *8th International Conference on Preservation of Digital Objects (iPRES)*, 2011.

[15] I. Santana-Perez and M. S. Pérez-Hernández, “Towards reproducibility in scientific workflows: An infrastructure-based approach,” *Scientific Programming*, 2015.

[16] https://github.com/wf4ever/ro-manager, 2016, [Accessed 02-May-2017].

[17] Standard OASIS, “Topology and orchestration specification for cloud applications version 1.0,” 2013.

[18] R. Qasha, J. Cała, and P. Watson, “A framework for scientific workflow reproducibility in the cloud,” in *IEEE e-Science*, 2016.

[19] P. J. Guo and D. Engler, “CDE: Using system call interposition to automatically create portable software packages,” in *USENIX*, 2011.

[20] P. J. Guo, “CDE: Run any Linux application on-demand without installation,” in *LISA*, 2011.

[21] Q. Pham, T. Malik, and I. Foster, “Using Provenance for Repeatability,” in *TaPP*, 2013.

[22] D. Thain, P. Ivie, and H. Meng, “Techniques for preserving scientific software executions: Preserve the mess or encourage cleanliness?” in *Proceedings of the 12th International Conference on Digital Preservation, iPRES 2015, Chapel Hill, North Carolina, USA, November 2-6, 2015*, 2015.

[23] Docker, "Docker," https://www.docker.com/, 2017, [Online; accessed 2-May-2017].

[24] Vagrant, "Vagramt," https://www.vagrantup.com/, 2017, [Online; accessed 2-May-2017].

[25] H. Meng, R. Kommineni, Q. Pham, R. Gardner, T. Malik, and D. Thain, "An invariant framework for conducting reproducible computational science," *Journal of Computational Science*, vol. 9, 2015.

[26] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *ACM SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, 2001.

[27] L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire, "noworkflow: capturing and analyzing provenance of scripts," in *International Provenance and Annotation Workshop.* Springer, 2014, pp. 71–83.

[28] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. I. Seltzer, "Provenance-aware storage systems." in *USENIX Annual Technical Conference, General Track*, 2006, pp. 43–56.

[29] A. Gehani and D. Tariq, "Spade: support for provenance auditing in distributed environments," in *Proceedings of the 13th International Middleware Conference.* Springer-Verlag New York, Inc., 2012, pp. 101–120.

[30] Q. Pham, T. Malik, B. Glavic, and I. Foster, "LDV: Light-weight database virtualization," in *ICDE*, 2015.

[31] Q. Pham, "A framework for reproducible computational research," Ph.D. dissertation, Dept. of Computer Science, University of Chicago, 2014.

[32] S. Dey, K. Belhajjame, D. Koop, M. Raul, and B. Ludäscher, "Linking prospective and retrospective provenance in scripts," in *Proceedings of the 7th USENIX Conference on Theory and Practice of Provenance*, ser. TaPP'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 11–11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2814579.2814590

[33] T. M. McPhillips, T. Song, T. Kolisnik, S. Aulenbach, K. Belhajjame, K. Bocinsky, Y. Cao, F. Chirigati, S. C. Dey, J. Freire, D. N. Huntzinger, C. Jones, D. Koop, P. Missier, M. Schildhauer, C. R. Schwalm, Y. Wei, J. Cheney, M. Bieda, and B. Ludäscher, "Yesworkflow: A user-oriented, language-independent tool for recovering workflow information from scripts," *CoRR*, 2015.

[34] P. Macko, D. Margo, and M. Seltzer, "Local clustering in provenance graphs," in *ACM CIKM*, 2013.

[35] Y. Tian, R. A. Hankins, and J. M. Patel, "Efficient aggregation for graph summarization," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: ACM, 2008, pp. 567–580. [Online]. Available: http://doi.acm.org/10.1145/1376616.1376675

[36] S. Cohen, S. Cohen-Boulakia, and S. Davidson, "Towards a model of provenance and user views in scientific workflows," in *Data Integration in the Life Sciences*, 2006.

[37] D. El-Jaick, M. Mattoso, and A. A. B. Lima, "Sgprov: Summarization mechanism for multiple provenance graphs," *JIDM*, vol. 5, no. 1, pp. 16–27, 2014.

[38] "The Sciunit," https://sciunit.run/, 2017, [Online; accessed 10-Sep-2017].

[39] City of Chicago, "Food Inspection Evaluation," https://chicago.github.io/food-inspections-evaluation/, 2017, [Online; accessed 05-2017].

[40] ——, "Chicago data portal," https://data.cityofchicago.org/, 2017, [Online; accessed 7-May-2017].

[41] ——, "Food Inspection Evaluation predictions-source code," https://github.com/Chicago/food-inspections-evaluation, 2016, [Online; accessed 7-May-2017].

[42] ——, "Food Inspection Evaluation," https://chicago.github.io/food-inspections-evaluation/predictions/, 2017, [Accessed 05-2017].

[43] J. Goecks, A. Nekrutenko, and J. Taylor, "Galaxy: A comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences," *Genome Biology*, vol. 11, 2010.

[44] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde, "Swift: Fast, reliable, loosely coupled parallel computation," in *IEEE Congress on Services*, July 2007, pp. 199–206.

[45] I. Altintas, O. Barney, and E. Jaeger-Frank, "Provenance collection support in the kepler scientific workflow system," in *IPAW*, 2006.

[46] Hydroshare.com, "Hydroshare," https://www.hydroshare.org/, 2017, [Online; accessed 2-May-2017].

[47] B. D. McKay, "Practical graph isomorphism," 1981.

[48] ——, "The page nauty," http://users.cecs.anu.edu.au/~bdm/nauty/, 2017, [Online; accessed 10-Sep-2017].

[49] Wolfram, “Find graph isomorphism,” http://reference.wolfram.com/language/ref/FindGraphIsomorphism.html, 2017, [Online; accessed 10-Sep-2017].

[50] M. M. Billah, J. L. Goodall *et al.*, “Using a data grid to automate data preparation pipelines required for regional-scale hydrologic modeling,” *Environmental Modelling & Software*, vol. 78, 2016.

[51] M. O. Rabin *et al.*, *Fingerprinting by Random Polynomials.* Center for Research in Computing Techn., Aiken Computation Lab, Univ., 1981.

[52] Tensorflow.org, “Tensorflow,” https://www.tensorflow.org/, 2017, [Online; accessed 2-May-2017].

[53] J. Freire, P. Bonnet, and D. Shasha, “Computational reproducibility: State-of-the-art, challenges, and database research opportunities,” in *ACM SIGMOD*, 2012.

[54] V. Stodden, P. Guo, and Z. Ma, “Toward reproducible computational research: An empirical analysis of data and code policy adoption by journals,” in *PloS one*, vol. 8, 2013.

[55] D. Penny, “Nature Reproducibility survey,” May 2016. [Online]. Available: https://figshare.com/articles/Nature_Reproducibility_survey/3394951

[56] T. Malik, “GeotrustHub,” https://geotrusthub.org/, 2017, [Online; accessed 10-Sep-2017].

[57] X. Ma, S. E. Beaulieu, L. Fu, P. Fox, M. Di Stefano, and P. West, *Documenting Provenance for Reproducible Marine Ecosystem Assessment in Open Science.* IGI Global, 2017, ch. 5, pp. 100–126. [Online]. Available: http://dx.doi.org/10.4018/978-1-5225-0700-0.ch005