Techniques for Preserving Scientific Software Executions: Preserve the Mess or Encourage Cleanliness?

Douglas Thain, Peter Ivie, and Haiyan Meng
Department of Computer Science and Engineering
University of Notre Dame
{dthain|pivie|hmeng}@nd.edu

ABSTRACT

An increasing amount of scientific work is performed *in silico*, such that the entire process of investigation, from experiment to publication, is performed by computer. Unfortunately, this has made the problem of scientific reproducibility even harder, due to the complexity and imprecision of specifying and recreating the computing environments needed to run a given piece of software. Here, we consider from a high level what techniques and technologies must be put in place to allow for the accurate preservation of the execution of software. We assume that there exists a suitable digital archive for storing digital objects; what is missing are frameworks for precisely specifying, assembling, and executing software with all of its dependencies. We discuss the fundamental problems of managing implicit dependencies and outline two broad approaches: preserving the mess, and encouraging cleanliness. We introduce three prototype tools for preserving software executions: Parrot, Umbrella, and Prune.

General Terms

Frameworks for digital preservation

Keywords

software preservation, dependency management

1. INTRODUCTION

While it has long been common for scientific publications to be prepared via computer, today much scientific work is now done completely from beginning to end in a computer. An elaborate model system may be run in simulation, generating raw data which is then processed by complex analysis software, which produces outputs that are displayed by visualization software, which can then be included in a final publication for dissemination and peer review.

Much early work in digital preservation from the library community focused on preserving the final artifact of that chain of effort: the publication. This includes accounting for physical media decay and obsolescence in addition to ensuring the availability of software for interpreting the data so it can be displayed to a user[24]. However, scientific productivity and integrity depends significantly upon our ability to preserve, share, and use the earlier steps in that chain, including both the software and the data. A peer-reviewer might wish to delve into the data associated with a paper, beyond the summary graph presented by the author. A collaborator might wish to pick up the current experimental software stack and adjust some parameters in order to obtain a new result. A competitor might wish to evaluate a completely new technique and compare it with a published technique in order to ensure that the previous technique has been validly recreated.

Unfortunately, the current state of the art is not encouraging. For example, in the biotech industry, Amgen attempted to reproduce 53 "landmark" articles in cancer research. They only succeeded with 10% of them [3]. In pharmaceuticals, Bayer was only able to reproduce about 21% of published results in 67 different projects [22]. Other efforts [25] have pointed out that there is a clear gap between preservation policies and practices. One can easily see why: a published computational result may briefly state that it ran with a certain version of software on a certain operating system, but may fail to state critical configuration values, dependent software, or even the precise inputs to the program. It is a common tale, even in the field of computer science, that an experiment was not published with enough details to accurately verify the results.

In this paper, we consider from a high level what techniques and technologies must be put in place to allow for the accurate preservation of the execution of software. We assume that there exists a suitable digital archive which can preserve digital objects for the long term, as are now commonly in place at university libraries, academic publishers, and so forth. The challenge lies in precisely identifying what must be preserved, naming each object appropriately, and providing a means for the consuming user to reassemble and verify the result.

The fundamental challenge throughout is the matter of **implicit dependencies.** In our current systems, it is all too easy for the user of a computer to consume some resource (a file, a program, a web site) without explicit knowledge that they are doing so. This leads us to two broad ap-

iPres 2015 conference proceedings will be made available under a Creative Commons license.
With the exception of any logos, emblems, trademarks or other nominated third-party images/text, this work is available for re-use under a Creative Commons Attribution 3.0 unported license. Authorship of this work must be attributed. View a copy of this licence at http://creativecommons.org/licenses/by/3.0/legalcode.

proaches to software preservation: **Preserving the mess** involves allowing the user to keep working in the current way while supplementary tools identify dependencies automatically. **Encouraging cleanliness** requires the user to state more clearly in advance what they are attempting to do. As we will show below, preserving the mess is easy but results in preserved objects that are of little use beyond identical verification, while encouraging cleanliness is harder but encourages extension and comparison.

Along the way, we give an overview of three pieces of software that demonstrate some of these approaches to software preservation. **Parrot** [16] enables the end user to preserve a mess by automatically capturing the file and network dependencies that form the environment of an application. **Umbrella** [17] encourages cleanliness by providing a precise way to specify and instantiate a software execution environment. **Prune** goes further by tracking and recording a software execution in the form of of individual operations that build upon each other's outputs. Each of these prototypes has been developed in the context of an NSF-supported project, called Data and Software Preservation for Open Science (DASPOS), which is examining the needs of preservation for the high energy physics community.[1]

2. SIMPLIFIED EXAMPLE

We consider the following simplified example in order to define some terms and highlight preservation challenges that we have encountered in working with a variety of applications. Suppose that a user has a laptop running GreenSock Linux 8.3 and wishes to run an open source simulation program `mysim` 3.2 on a custom input file `data` to produce a single output file `result` by typing the following command into the terminal:

```
$ mysim -in data -out result
```

The user's objective is now to preserve not just the software itself, but that specific *execution* of the software, so that others can verify a result and also extend and compare it to new methods.

The diligent but naive user might attempt to preserve this particular execution by saving the input file `data` in a digital repository, then making a note of the unique identifier of the data, and the exact version of `mysim` used in the published paper. In principle, the reader of the paper must simply install the given version of the software, download the data, and will quickly be able to verify, extend and compare with the published work. Unfortunately, this procedure is insufficient. The main problem is that what is visible to the user is only the tip of the iceberg in terms of what is necessary to actually execute the program.

Figure 1 gives a better sense of what may be involved in preserving such an execution. The binary executable `mysim` obviously depends upon `data` as an input file, but perhaps it also reads a file of calibration data `calib` in the current directory which is not mentioned on the command line, but hard-coded into the program. Further, the executable program itself does not stand alone, but depends upon a specialized library `libsim` that the user had to install onto the machine at some point. What's worse, the program itself makes a network connection at runtime in order to download some critical configuration data `config` from a public web server.

[1] `http://www.daspos.org`

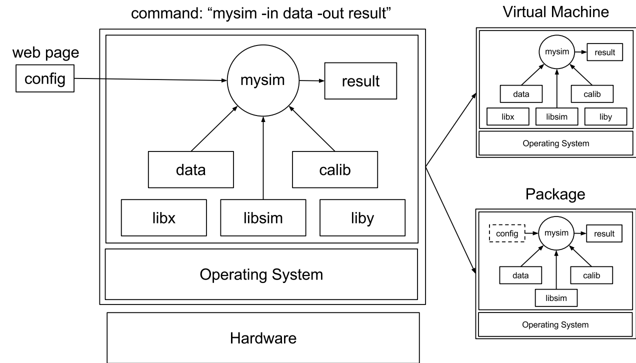

Figure 1: Preserving Implicit Dependencies
Even the simplest of programs has both explicit and implicit dependencies. Saving the program in a virtual machine tends to capture unnecessary items, while automatic packaging can identify exactly what objects are needed at runtime.

Even that is not the whole story. Even the simplest software depends on a complex stack of objects present on the local machine, including libraries, scripts, configuration files, and the operating system kernel. Together, these comprise what we call the **environment** of the program. While these components are of course required for the software to run, they are not the primary interest of the user, who cares first and foremost about the simulation and the data. In principle, the simulation should run correctly and yield the same results when run on a different (but compatible) operating system. In practice, it might not, and so preserving the environment is necessary for long-term viability.

Effective preservation requires that there exist some form of hardware capable of running the operating system and software. This could be physical preservation of a hardware artifact, or a compatible virtual implementation. Hardware preservation makes it easy to reproduce an application, but is not efficient due to the cost and space overhead for maintaining old hardware. At some point, the preserved hardware may become completely unusable due to humidity or the lifetime of components like disks. A compatible virtual implementation of old hardware, such as Olive [26], recreates the original execution environment on the future platforms through virtualization techniques [23].

An additional complication is that the different layers of the system may be provisioned by different parties. In the case of a personal laptop, the same person purchased the hardware, installed the operating system, and ran the software. But in a complex university computing environment, the hardware procurement, operating system installation, and software deployment may all be accomplished by multiple teams of people. By the time the end user gets involved, they may have no idea what the underlying environment

actually contains!

The essence of the software preservation problem is that it is extremely difficult for the end user to understand the set of objects upon which an execution depends. The visible user interface suggests that the only required components are `mysim` and `data`, but the reality is that the program cannot run without a complex and interdependent set of invisible objects. Unless some additional specification or restrictions are put in place, any file on the local filesystem or any service available on the Internet could be a potential dependency of the execution. A preservation solution must either automatically capture what is unseen ("capturing the mess") or structure the user's interactions to make all dependencies explicit ("encourage cleanliness").

3. PRESERVATION OBJECTIVES

(Many terms are used in the field of digital preservation, including reproducibility, re-use, re-creation, re-purposing, and more, each with slight variations in meaning. To avoid confusion, we limit our terms to **preservation** to denote digital preservation whose purpose includes **verification** of previous results and **extension** to new results.)

Before posing solutions, it is useful to consider how the preserved software execution may be re-used in the future. It is commonly stated that researchers wish to precisely reproduce other's work so as to verify the truth of published claims [20]. In discussing the matter with a variety of researchers, we have found little appetite for attempting to prove or disprove other's work in this way. Rather, there are a wide variety of other motivations for precise reproducibility, most of them in the realm of reducing the amount of labor required to continue forward from a previous result. Examples include:

- **Identical Verification.** The same software executes on the same input data in the same environment and is repeated to verify that it produces the same result. This is done to evaluate the soundness of the reproduction system itself before moving on to other matters.
- **New Environment Verification.** The same software executes on the same input data in a **new** environment to verify that it produces the same result. This approach is taken to evaluate the soundness of new libraries, operating systems, hardware, and other parts of the environment as they evolve independently of the scientific objectives.
- **New Software Verification.** A new version of the same software executes on the same input data in the same environment, so as to verify that an improved implementation of the same algorithm yields the same results as the old.
- **Extension to New Data.** The same software executes on **new** data in the same environment. This allows previously published techniques to be extended to new data sets with confidence that new results are not affected by changes to the software or environment.
- **Extension to New Software.** Completely different software executes on the same data in the same environment. This allows for the direct comparison of different or competing algorithms on identical data, with confidence that the new publication has accurately reproduced the competing result.

Each of these use cases (except the first) requires a clear separation between the scientific software, the scientific data, and the computing environment, so that each can be evolved independently without accidentally modifying the other.

4. PRESERVING THE MESS

We first consider "preserving the mess" approaches, in which we attempt to capture exactly what the user attempted, without interfering in the setup of their work.

4.1 Virtual Machine Technology

A commonly-proposed solution is that software executions should be preserved by placing the software, data, and environment within a single virtual machine, then preserving the machine image in a repository either before or after the execution. This is a relatively easy technique for the user to apply, as long as the boundaries of the application correspond to the boundaries of a single machine and filesystem. If the application only depends upon objects in the local filesystem, each will be preserved at the bit-level in a precise way. Virtual machine preservation is effective and is already being used today at a small scale to capture individual complex systems [12].

However, when we consider preserving a large number of results that may evolve over time, virtual machine preservation has some significant limitations:

- **Imprecise Capture.** A virtual machine image will almost certainly contain items that are irrelevant to the task at hand. For example, a standard operating system contains a wide variety of software to handle many different user needs, most of which are not used by a given execution. Even worse, if the user preserves the image of their personal laptop, it could be all to easy to accidentally preserve personal data or legally sensitive information. On the other hand, the machine image by itself may fail to capture external dependencies (such as the `config` file on the web server) that are not strictly within the image, causing re-use to fail if the external dependency is not present.
- **Rigid Composition.** A virtual machine image intermixes the various components of the system in ways that are difficult to undo automatically. Absent some additional specification, there is no automatic way to distinguish the inputs to the simulation from the files comprising the application or the operating system. Manual effort to browse the image is the only way by which items can be extracted from the machine image.
- **Inefficient Storage.** It is rare for a single software execution to have scientific validity on its own. Rather, it is common for a researcher to run thousands to millions of instances of an application on a high-throughput computing system, each one using a slightly different input file or parameters. If we attempt to preserve each

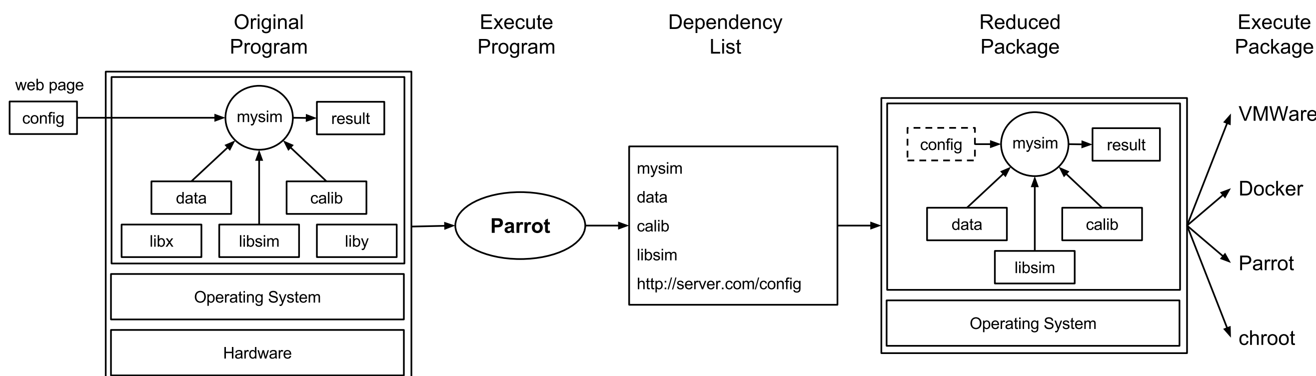

Figure 2: Packaging an Application with Parrot
Parrot can be used to trace the files and network objects used by a conventional program, and produce a listing of the items on which it depends. This listing is used to create a reduced package that can be re-executed by multiple technologies.

instance of the application in its own virtual machine, an enormous amount of storage will be consumed by duplicating the software, environment, and other components that are common to each instance.

- **Huge Image Size.** Data-intensive applications may have enormous input data sizes, measured in terabytes to petabytes. At this scale, the input data may be too large to store on a local disk, or to fit within a single virtual machine image. Large data sources are typically handled by purpose-built archives, and it is more effective for the virtual machine to refer to the archive than to duplicate its functionality.
- **Inefficient Execution.** There is sometimes an assumption that with a virtual machine "performance is of secondary importance" [14]. While this can be appropriate in some cases, users tend to stretch the limits of the available hardware to perform increasingly complex analyses. If a preservation method causes too much of a performance hit users will be unlikely to consider it until after getting their work done, if ever.

We conclude that the simple method of capturing a virtual machine image – while it may be useful – will not be an effective long-term strategy for preserving scientific software and data in a way that facilitates verification and extension.

4.2 Container Technology

Container technology is a growing alternative to hardware virtualization. Multiple containers can execute simultaneously on a single operating system kernel, and have lower execution overhead because they run directly on the CPU without translation or interception. Linux Containers (LXC), Rocket [1] and Docker [18] are examples of current systems that use this technology.

The stored image of a container is merely a stored filesystem tree. it may be stored as a disk image for efficiency, but can easily be exported in a portable, shareable format such as `tar` or `zip`. A container image can be a large, completely functional operating system with multiple applications, but users of container systems are encouraged to make small, minimal container images that support a single application at a time. However, the user must have enough understanding of the underlying application in order to construct the minimal image.

Although containers differ from virtual machines in the technology of execution, the container images themselves have the same problems as saving a filesystem image in a virtual machine, specifically imprecise capture, rigid composition, and inefficient storage. To use either technology effectively, the user needs additional help to identify dependencies.

4.3 Package Reduction with Parrot

Tracing techniques can be used to determine the minimal set of objects needed to support an application, and then use that information to construct an appropriate package of actual dependencies in either a virtual machine image or container image. A monitor process can run alongside an executing instance of an application, observe its interactions with the environment, and then save only those elements of the environment into a new package. A variety of technologies can then be used to re-execute the software.

Parrot [16] is an example of this technique, which is also employed by CDE [10], and PTU [21]. Parrot was originally designed to be a remote filesystem access tool which connects conventional applications to remote I/O systems such as HTTP and FTP. It works by trapping system calls through the `ptrace` interface and replacing selected operations with remote accesses. Through this technique, Parrot is able to modify the filesystem namespace in arbitrary ways according to user needs. Parrot is particularly used in the high energy physics community to provide remote access to application software via the CVMFS [4] file system.

To support package creation, we made small modifications to Parrot to record the logical name of every file accessed by an application into an external dependency list. After execution is complete, a second tool is used to copy all of the named dependencies into a package. In addition, Parrot tracks the network operations of an application and the data passing through them. It records the address, port number, and protocol of each connection. In addition, it examines

each connection for known protocol signatures and can determine the protocol-level endpoint of the connection. For example, if the application connects to a webserver, Parrot can record not only the address of the webserver, but also the URL which the application accessed for common protocols such as HTTP, SVN, and GIT. (Parrot is limited in that it cannot inspect encrypted data, beyond indicating that a TLS/SSL connection was made.)

Figure 2 gives an overview of how a package is made. First, the user executes the program in the normal way, using Parrot. The application runs to completion while Parrot collects the files and URLs accessed into a **dependency list**. All the accessed files are copied into a package so that the file system structure (relevant paths between files, and symbolic links) is kept within the package. The package is a simple `tar` archive that can be recorded in any digital repository and then re-executed by a variety of techniques. For example, the package can be converted into a virtual machine image and executed by VMWare [23], or it can be converted into a container image and executed by Docker [18]. Parrot itself can also be used to re-execute the package by mounting the package directory as the application's root directory.

The reduced package is certainly smaller than the entire virtual machine image, but can still be astonishingly complicated. As shown in Figure 2, all the file dependencies, including files from the root filesystem like `/bin` and files from the network filesystems like AFS and CVMFS, are denoted as file paths within the dependency list. The distinction between input data and software is lost, which makes extensions based on a preserved package difficult. In addition, common library dependencies will be wrapped into different packages multiple times, which increases the storage overhead of the remote archive. In an earlier work, we used Parrot to preserve a simple high energy physics application called `TauRoast`. The reduced package contained 22,068 files and directories totaling 21 GB of data and software drawn from 8 different filesystems. Virtually all of this detail was unknown to the invoking user.

Based on this experience, we believe that these approaches are ultimately limited. While "preserving the mess" is better than not preserving at all, the resulting packages are extraordinarily complicated, and provide the end user with little traction for understanding the behavior well enough to extend the software. Preserving the mess is inherently retrospective – it involves observing an execution after it executes to infer what resources were consumed. A more structured approach is needed for extending the original work.

5. ENCOURAGING CLEANLINESS

In contrast to preserving the mess, "encouraging cleanliness" is a forward-looking approach. Cleanliness is accomplished by encouraging everyone to name and preserve objects **before** they are used, then to combine the objects at runtime in a way that clearly distinguishes the reusable layers of the application. To support cleanliness, an archive is needed to maintain the OS images, software, and data for each software execution. A specification should be created to describe the execution environment for each execution with the help of the system administrator and the original author.

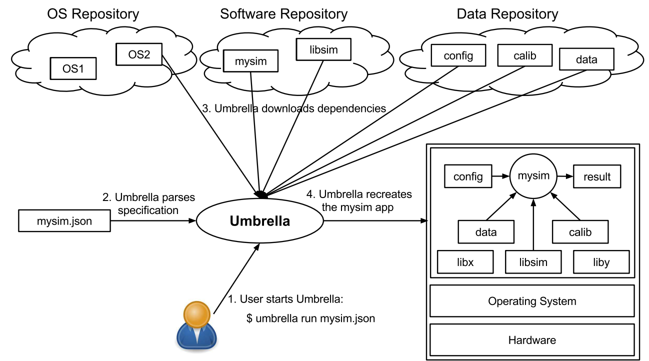

Figure 3: Overview of Umbrella
Umbrella is used to execute a specification of an application which describes precisely how the operating system, software, and data are combined at runtime.

Here, we demonstrate two approaches to cleanliness: Umbrella preserves the execution of a single software execution by precisely naming the hardware, operating system, software, and data necessary to carry it out. PRUNE preserves the execution of a workflow of software executions by preserving multiple software executions independently, then using Umbrella to execute each one precisely.

5.1 Precise Execution with Umbrella

Umbrella [17] is designed to enable the precise construction of an execution environment for software. Figure 3 gives an overview of the system. The user gives a declarative specification of the desired execution environment, encompassing the hardware, kernel, OS, software, data, and environment variables, without being tied down to a single virtualization technology. Umbrella considers each of the elements of the specification, downloads the files needed, constructs the complete environment by combining the components, then runs the program.

Figure 4 gives a possible Umbrella specification for our example program. The `hardware` section indicates the required CPU architecture, the CPU model, the CPU flags, the number of cores, and the amount of memory, disk and other hardware requirements. The `kernel` section defines the type and version of the operating system kernel, which may be a single value or a range. The `os` section provides the name and version information of the operating system, which includes the system software in the root filesystem, apart from the kernel. The `software` section provides the software name, version, and platform of each required software package. The `data` section indicates the necessary data dependencies, and their mount points. The `environ` section sets the environment variables for an application. For each category, a variety of methods of naming the object are available, ranging from **unique identifiers** (`id="e5f3cd"`) to **abstract attribute values** (`version="6.5"`), depending on what is most appropriate for the user. The user may select whichever method best meets their needs. We discuss tradeoffs in naming schemes at length below.

Note that Umbrella requires the user to be explicit about external dependencies. As our example shows, the exter-

```
"hardware": {
    "platform": "x86_64",
    "cpu cores": "1",
    "memory": "1 GB",
    "disk": "4 GB"
},
"kernel": {
    "type": "Linux",
    "release": "2.6.32"
},
"os": {
    "name": "GreenSock",
    "version": "8.3"
},
"software": {
    "mysim": {
        "id": "f6e17cc80...",
        "mountpoint": "/software/mysim",
        "version" : "3.2"
    }
},
"data": {
    "config": {
        "url" : "http://server.com/config",
        "mountpoint": "/etc/mysim/config"
    },
    "data": {
        "id": "cb9878132...",
        "mountpoint": "/home/test_user/mysim/data"
        }
    },
},
"environ": {
    "HOME": "/home/test_user",
    "PATH": "/usr/bin:/software/mysim/bin"
},
"command": "mysim -in data -out result"
```

Figure 4: Example Umbrella Specification
This example of an Umbrella specification indicates exactly how the components of `mysim` *come together to form a complete execution.*

nal web page containing `config` is explicitly mentioned, so that Umbrella itself will download the data and provide it to the application. The user may make a value judgement about the long-term availability of the external dependency. To avoid data loss, `config` should be archived into the data repository, together with its metadata including its checksum, size, authorship, access permission and usage. The specification of `mysim` should include `config` as one of its data dependencies through its unique identifier or attribute list. Similarly, the stability and persistency of all the third-party dependencies should be evaluated, and the unstable ones should be ingested into the archive if access permission is allowed. Once all items are archived, then the specification itself is a (compact) archivable object that completely describes the execution.

The Umbrella specification is deliberately silent about the specific *mechanism* by which the program will be re-executed. This gives the implementation freedom to make use of new technologies as they are developed, or to harness whatever resources are available at the moment of execution. For example, if Umbrella is invoked on a machine that already has the desired operating system on compatible hardware, then it can simply run the software directly. If the hardware is compatible but the OS is not, then Umbrella can attempt to use a container to deploy the desired OS. If not even the hardware is compatible, then Umbrella can instantiate a virtual machine or contact a commercial cloud service to create the desired environment.

The specification is inherently efficient in both use of storage space, and in construction of the desired environment. Each of the components in the specification is assumed to be preserved in an external digital repository, then downloaded and cached at the execution site as needed. Obviously, if multiple executions use the same operating system or the same dataset, it is only necessary to keep one copy in the archive and share it at runtime among multiple executions.

Previous approaches to the provisioning of virtual machines, such as V-MCS [28], FutureGrid [30], Grid'5000 [6], and VMPlants [13], achieve various environments by applying executable scripts to base virtual machines. While effective, this can be quite slow while data is copied or updated in place. In contrast, Umbrella *mounts* each object in the filesystem namespace, so that at runtime, the collection of objects is effectively instantaneous.

5.2 Preserving Workflows with PRUNE

While Umbrella describes how to precisely perform a *single* software execution, PRUNE describes how to connect *multiple* executions together, such that entire workflows can be preserved, verified and extended. The key idea of PRUNE is to represent every invocation of a program as the evaluation of a function on immutable digital objects. In PRUNE, our example program would be invoked as a function call:

```
Result = MySim( Config, Data )
```

In this example, `Config` and `Data` refer to data items stored in "PRUNE-space", a local repository of immutable objects. `mysim` consists of an Umbrella specification of how to execute the program in a precise environment, while `Result` refers to the output file, which is moved into PRUNE-space when the program completes.

Over time, as the user runs a large number of programs, they conceptually build up a large graph of objects, each related to each other by function invocations. If an object was created by a chained series of function calls, PRUNE retains enough information to accurately describe the steps necessary to create that object from beginning to end. The user who wishes to publish a paper depending upon a result can ask PRUNE to produce a package containing every dependency needed for that result, which can then be archived along with the scientific publication.

Of course, accumulating those objects over time will exhaust disk storage, or the user's budget for cloud storage. To this end, PRUNE can safely delete the binary form of any object in PRUNE-space, because it retains enough information to re-create it, should the user require that it be produced. In this way, storage costs can be traded for computation costs as needed.

PRUNE gives all objects a uuid, but managing a large number uuids manually would quickly become cumbersome. So each repository in PRUNE has it's own namespace on top of the uuids such that a name points to a uuid, and both the name and uuid are preserved. A collaborating reposi-

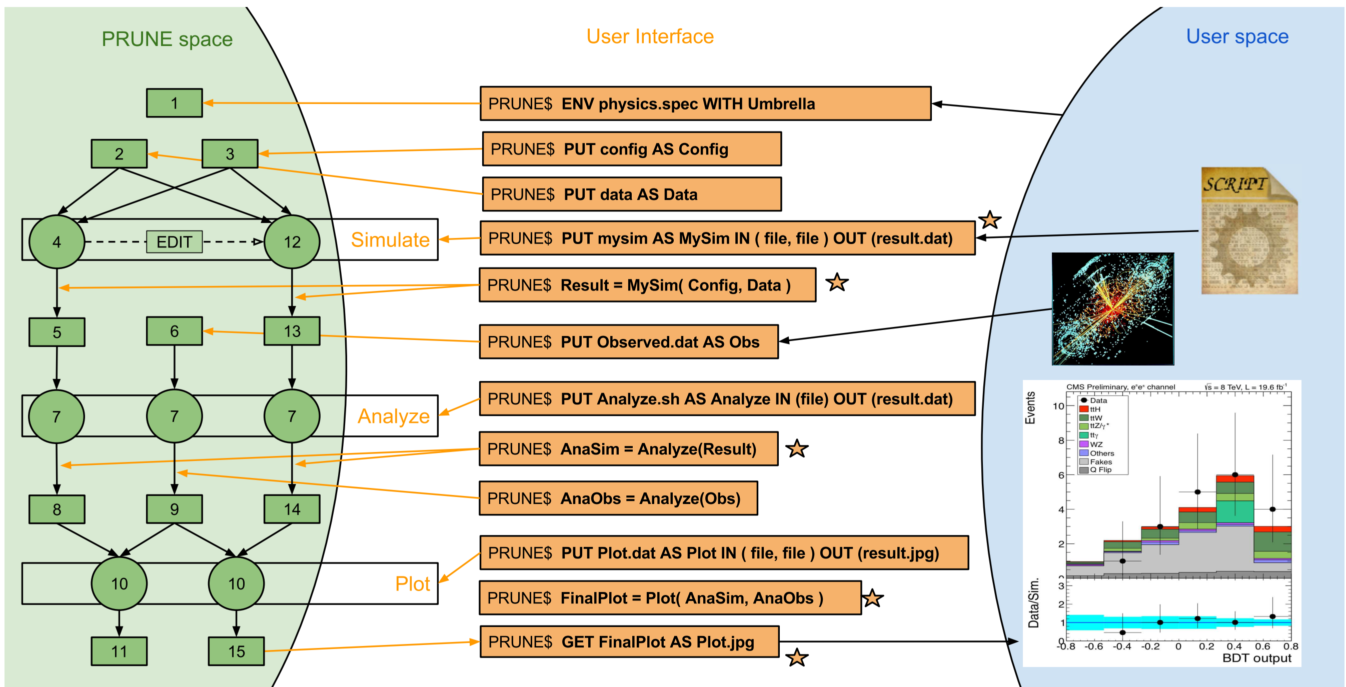

Figure 5: PRUNE overview

Prune represents each invocation of a program as a function call on immutable archived objects. As the user invokes more and more functions, a tree of archived objects accumulates. Each execution is made precisely reproducible via Umbrella.

tory can choose to use the same name, or not, but uuids are immutable across repositories.

In PRUNE, a distinction is made between *operations* which are specified programmatically, and *edits* that are transformations performed manually and might not even have a detailed description. Precise reproducibility is possible in all cases, but including an edit in the workflow could leave a gap in the provenance and make it difficult for a collaborator to reproduce an edit on a file if the original file has changed.

An edit which does not leave a gap in the provenance is shown in Figure 5 from object 4 to 12. This edit allows to user to easily manage minor evolutionary changes to the workflow without harming the ability to preserve the workflow for collaborators. Notice that tracing 11 or 15 back to their original source files does not require passing through the edit.

6. PRESERVATION CONSIDERATIONS

Within the overall strategies outlined above, a number of tradeoffs become apparent between user effort, preservation cost and complexity, and the generality of the artifacts for re-use over the long term. Here we give some overview of these tradeoffs and suggest when one approach or the other (or both) may be appropriate.

6.1 Source vs. Binary Code

Should we preserve the source form or the binary form of compiled programs?

By design, source code in a high level language such as C is designed for human consumption and is the preferred form for understanding and modifying the program. The binary form produced by the compiler can be directly executed but is of little use for analysis and may not function in even a slightly different environment. Source code is obviously the preferred form in which software itself achieves longevity as an independent entity.

However, if our goal is to preserve an *instance of executed software*, the answer is not so clear. If only the source code is preserved, then re-use requires that a suitable compiler, linker, and other supporting tools be present at re-use time. Languages are not always forward compatible, which requires us to preserve the actual compiler used in addition to the software. Moreover, the source code for the compiler also needs to be preserved, and so on. As the dependency chain increases, the cost of re-execution increases, as well as the risk of failing to build a usable binary.

Thus, the comprehensive approach is to preserve both the source code and the resulting binary, so that they are mutually verifiable. If the surrounding environment is faithfully preserved, then the binary will remain usable. If it is desired to rebuild the software from source, the correctness of the rebuild can be confirmed by comparing its outputs against those generated by the preserved binary. Similar arguments apply to any complex object constructed from text instructions, such as an RPM package from a rpmbuild script or a Docker image from a Dockerfile.

6.2 Manual vs. Automatic Preservation

Should preservation be performed automatically, or only at the user's request?

Automatic preservation does not need lots of involvement from the user, but can be very messy. Since everything is recorded, irrelevant operations become part of the preserved data, which are difficult to distinguish from the relevant operations. The irrelevant operations may include listing files in a directory or iterations that failed to produce the desired results and had to be modified and re-run. Operations at lower levels may be difficult to decipher for even the original researcher. It is nearly impossible for another researcher to use this type of data to do extension work.

Automatic preservation may also cause privacy issues for the researcher. A preservation tool which is allowed to track the software execution may preserve the researcher's ssh private key and private key file including his Amazon EC2 key pair. Distributing the preserved execution may leak the private information of the researcher to unintended targets.

At the other extreme, manual preservation places the entire burden of reproducibility on the researcher. The researcher might create a script that includes the final list of operations they used to produce the experimental results. Or the researcher might create documentations explaining each step with details about why certain decisions were made. Or the user might include very little information about how the results were obtained, making the preservation ineffective.

6.3 Pre vs. Post Preservation

Should the burden of preservation come before or after the user's work is completed?

Most researchers choose to wait until they have their full results before preserving their methods. Unfortunately, by the time the results are available, other factors come into play which make this approach unlikely to succeed. There is little motivation to put in the extra effort to identify how results were obtained since it does not appear to be a factor in whether or not a paper is accepted. The researcher typically gets busier as a deadline approaches and preservation has a low priority. In addition, sometimes important details are forgotten or only known by system administrators. In a collaborative effort, students who were involved may have moved on by that time. Also, the original execution environment may have changed, and the original operations no longer work.

An alternative approach is to require the researcher to preserve every step along the way before it is even executed. This would require more work for the researcher up-front, but has a much higher likelihood of resulting in a preserved software execution. However, this approach would also include failed attempts or extraneous commands that occur as the research evolves. This extra data puts additional load on resources used to capture and store the information.

A middle ground can be found by provisionally preserving everything in a local repository, and then enabling the user to identify (at a later time) what objects should be retained permanently. This requires more effort from the researcher both before and after the execution, but provides a clean description of how the research can be verified, extended and compared. Furthermore, the extra burden on the user might be offset by providing some additional tools with convenience features.

6.4 Unique Identifiers vs. Attributes

How should preserved software components be identified?

Each component of a software execution should allow the user to refer and verify its integrity. There are two broad approaches for this: **unique identifiers** or **attribute descriptions**. The following are examples of each type.

```
Unique Identifiers:
    doi = "10.7274/R0C24TCG";
    checksum = "f6e17cc80...";
    url = "http://server.com/config";

Attribute Descriptions:
    name = "mysim";
    version = "3.2";
    architecture = "x86_64";
```

Unique identifiers provide an unmistakable reference to a single binary object. A Digital Object Identifier (DOI) [19] is an example that names a publisher, then an object, which can be resolved by the Handle System [27] to a current location of the desired object, in the form of a Uniform Resource Locator (URL). DOIs are widely used by digital libraries to identify published documents, and to a lesser extent, other kinds of digital objects. However, the DOI system recommends, but does not force, the objects referred by a DOI name to be persistent or immutable. For example, the DOI name `10.1000/182` always refers to the *latest* version of the DOI handbook, which is the primary source of information about the DOI system. [2]

Attribute descriptions describe essential properties of the software but do not necessarily uniquely identify a image. A suitable set of attribute-value pairs can be used to search a known repository for corresponding images that satisfy the given requirements, and are likely to resolve to a small set of compatible objects.

It seems that despite all of the advances that an internet connected research community provides, the question of archiving identification information is an issue yet to be resolved [29]. As long as the unique identifiers are kept immutable, they could be used as persistent identifiers in a global system of identifiers [2]. And a similar system focused on the evolving translations could provide both user friendliness and reproducibility.

[2]As an aside, the DOI infrastructure is almost, but not quite, suitable for naming software components. The main problem is that DOIs generally resolve to a web page that describes the "concept" of the object for a human reader, rather than resolving to the binary object itself. What is needed is a unique name that resolves directly to the object, or a convention for resolving the object from the concept page itself.

7. RELATED WORK

The problem of software verification is not new. As far back as 1984 [5], efforts were made to encourage verification that design specifications matched the actual behavior of software. However, the paper demonstrated that when the guidelines are followed, deviations from the specifications can be detected earlier, saving time in the overall software development. This tight coupling of specification and implementation has benefits for preservation also.

Dendro [8] has the user do some work as early as possible in order to preserve the provenance. For example, rather than just providing a link to a website, a triple is used to describe that the URL is the creator's web page. Using this ontology based data model rather than relying on a relational database, the preservation becomes self-documenting. A system called dataref versuchung [9] also requires some upfront work by the researcher. But once the researcher has properly created a figure for a LaTeX document, the system automatically includes a datagraphy that includes information about how the figures were created.

Unlike the above approaches which require significant user intervention, the PERICLES Extraction Tool [7] is initiated at runtime and attempts to automatically detect all implicit dependencies on the system environment and convert them to explicit dependencies. It also attempts to rank significant events to make the result more organized. However, it is still possible that this tool may miss implicit dependencies introduced by the extension work.

Matthews [15] proposes a conceptual framework for software preservation which includes a performance model of software and its input data, a model of software components, and the categories of preservation properties of software such as functionality, composition, provenance, ownership, execution environment and so on. Hong [11] proposes a benefits framework for software preservation which enumerates different purposes of software preservation and its benefits, analyzes the pros and cons of integrating software preservation measures into software development processes and preserving legacy software separately, and provides different options for software preservation. In contrast to this research on software preservation, our work considers the preservation of scientific software executions systematically, which includes data, software, and execution environments.

8. OPEN PROBLEMS

In this paper, we have outlined what we see as the most pressing problems of digital preservation and outlined broad strategies for solving them. There remain many hard problems to consider:

Preservation of Distributed Applications. Compared with single-machine applications, the preservation of distributed applications is more challenging due to the following facts: First, a distributed system is often composed of multiple computer nodes, each has its own software stack. Second, the distributed model and the network configuration must be maintained to reconstruct the distributed systems. Third, some distributed systems like HTCondor are dynamic, in that nodes can join and leave the Condor pool at any time. Should we preserve distributed applications including software and hardware completely? Should we just preserve the detailed configuration of distributed applications? Should we only preserve the working principle of distributed applications?

Preservation Granularity. To preserve applications, there are three different kinds of packages involved in an archive: Submission Information Packages (SIPs), Archive Information Packages (AIPs) and Dissemination Information Packages (DIPs) [31]. The granularity for these packages may be different. For example, an archive may choose to split submission packages into smaller pieces to fit its underlying storage architecture. The choice of granularity depends on the overhead of metadata management, storage overhead, the time overhead of submission, storage and reconstruction, and user-friendliness.

Preserving Preservation Tools. Emulation, as an important preservation approach, emulates the original execution environment of an application to allow the application to execute without modification. Compared with migration, where every preserved application needs to be somehow modified to fit the new environment, emulation keeps all the applications unchanged, and emulates the previous execution environment [14, 24].

Commercial Software and Sensitive Data. Preserving both the source code and binary code can help the new users extend the work easily. However, sometimes it is difficult to get the source code of software, especially commercial software. The preservation policy for this type of software must take copying and distribution conditions into consideration. On the other hand, trapping system calls may expose some sensitive data, which should require special access permissions. Before wrapping all these data into a reduced package, the sensitivity of the preserved data should be considered.

Acknowledgements

This work was supported in part by National Science Foundation grant PHY-1247316, Data and Software Preservation for Open Science (DASPOS).

9. REFERENCES

[1] CoreOS is building a container runtime, Rocket. `https://coreos.com/blog/rocket/`, 2015.

[2] B. Bazzanella. A persistent identifier e-infrastructure. *IPRES 2014 proceedings*, page 118, 2014.

[3] C. G. Begley and L. M. Ellis. Drug development: Raise standards for preclinical cancer research. *Nature*, 483(7391):531–533, 2012.

[4] J. Blomer, P. Buncic, and T. Fuhrmann. CernVM-FS: delivering scientific software to globally distributed computing resources. In *Proceedings of the first international workshop on Network-aware data management*, pages 49–56. ACM, 2011.

[5] B. W. Boehm. Verifying and validating software requirements and design specifications. In *IEEE software*. Citeseer, 1984.

[6] R. Bolze, F. Cappello, E. Caron, M. Daydé, F. Desprez, E. Jeannot, Y. Jégou, S. Lanteri, J. Leduc, N. Melab, et al. Grid'5000: a large scale and highly reconfigurable experimental grid testbed.

International Journal of High Performance Computing Applications, 20(4):481–494, 2006.

[7] F. Corubolo, A. Eggers, A. Hasan, M. Hedges, S. Waddington, and J. Ludwig. A pragmatic approach to significant environment information collection to support object reuse. *IPRES 2014 proceedings*, page 249, 2014.

[8] J. R. da Silva, J. A. Castro, C. Ribeiro, and J. C. Lopes. The dendro research data management platform. *IPRES 2014 proceedings*, page 189, 2014.

[9] C. Dietrich and D. Lohmann. The dataref versuchung: Saving time through better internal repeatability. *ACM SIGOPS Operating Systems Review*, 49(1):51–60, 2015.

[10] P. J. Guo and D. R. Engler. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *USENIX Annual Technical Conference*, 2011.

[11] N. C. Hong, S. Crouch, S. Hettrick, T. Parkinson, and M. Shreeve. Software preservation benefits framework. *Software Sustainability Institute Technical Report*, 2010.

[12] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Scientific workflow applications on Amazon EC2. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 59–66. IEEE, 2009.

[13] I. Krsul, A. Ganguly, J. Zhang, J. A. Fortes, and R. J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *Supercomputing, 2004. Proceedings of the ACM/IEEE SC2004 Conference*, pages 7–7. IEEE, 2004.

[14] R. A. Lorie and R. J. van Diessen. UVC: A universal virtual computer for long-term preservation of digital information. *IBM Res. rep. RJ*, 10338, 2005.

[15] B. Matthews, A. Shaon, J. Bicarregui, and C. Jones. A framework for software preservation. *International Journal of Digital Curation*, 5(1):91–105, 2010.

[16] 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*, 9:137–142, 2015.

[17] H. Meng and D. Thain. Umbrella: A portable environment creator for reproducible computing on clusters, clouds, and grids. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '15. ACM, 2015.

[18] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2, 2014.

[19] N. Paskin. Digital object identifier (DOI) system. *Encyclopedia of library and information sciences*, 3:1586–1592, 2008.

[20] R. D. Peng. Reproducible research in computational science. *Science (New York, Ny)*, 334(6060):1226, 2011.

[21] Q. Pham, T. Malik, and I. T. Foster. Using provenance for repeatability. In *USENIX NSDI Workshop on Theory and Practice of Provenance (TaPP)*, 2013.

[22] F. Prinz, T. Schlange, and K. Asadullah. Believe it or not: how much can we rely on published data on potential drug targets? *Nature reviews Drug discovery*, 10(9):712–712, 2011.

[23] M. Rosenblum. Vmware?s virtual platform? In *Proceedings of hot chips*, volume 1999, pages 185–196, 1999.

[24] J. Rothenberg. *Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation. A Report to the Council on Library and Information Resources.* ERIC, 1999.

[25] B. Sierman. The scape policy framework, maturity levels and the need for realistic preservation policies. *IPRES 2014 proceedings*, page 259, 2014.

[26] G. St Clair and D. Ryan. Olive: A digital archive for executable content. *Coalition for Networked Information*, 2011.

[27] S. Sun, L. Lannom, and B. Boesch. Handle system overview. Technical report, RFC 3650, November, 2003.

[28] X.-H. Sun, C. Du, H. Zou, Y. Chen, and P. Shukla. V-mcs: A configuration system for virtual machines. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–7. IEEE, 2009.

[29] H. Van de Sompel and A. Treloar. A perspective on archiving the scholarly web. *IPRES 2014 proceedings*, page 194, 2014.

[30] G. Von Laszewski, G. C. Fox, F. Wang, A. J. Younge, A. Kulshrestha, G. G. Pike, W. Smith, J. Voeckler, R. J. Figueiredo, J. Fortes, et al. Design of the futuregrid experiment management framework. In *Gateway computing environments workshop (GCE)*, pages 1–10, 2010.

[31] E. Zierau and N. Y. McGovern. Supporting the analysis and audit of collaborative oais?s using an outer oais-inner oais (oo-io) model. *Preservation (DDP)*, 9:5.