Reproducibility in Scientific Computing

PETER IVIE and DOUGLAS THAIN, University of Notre Dame

Reproducibility is widely considered to be an essential requirement of the scientific process. However, a number of serious concerns have been raised recently, questioning whether today's computational work is adequately reproducible. In principle, it should be possible to specify a computation to sufficient detail that anyone should be able to reproduce it exactly. But in practice, there are fundamental, technical, and social barriers to doing so. The many objectives and meanings of reproducibility are discussed within the context of scientific computing. Technical barriers to reproducibility are described, extant approaches surveyed, and open areas of research are identified.

CCS Concepts: • **Information systems** → **Data provenance**; *Uncertainty*; • **Social and professional topics** → **Software management**; **History of software**; Project and people management; **Computational science and engineering education**; • **Software and its engineering** → **Consistency**; **Software verification**; **Software usability**; *Domain specific languages*; **Software configuration management and version control systems**; **Software libraries and repositories**; **Software maintenance tools**; **Reusability**; **Software verification and validation**; **Software evolution**; **Software version control**; **Maintaining software**; *Virtual machines*; *File systems management*; *Petri nets*; *Ultra-large-scale systems*; *Interoperability*; *Data flow languages*; *Very high level languages*; *Abstract data types*; *Compilers*; *Graphical user interface languages*; *Unified Modeling Language (UML)*; *Application specific development environments*; *Software design tradeoffs*; *System administration*; *Collaboration in software development*; • **General and reference** → *Validation*; *Verification*; Surveys and overviews; Experimentation; • **Theory of computation** → *Pseudorandomness and derandomization*; • **Applied computing** → *Physical sciences and engineering*; Physics; • **Computer systems organization** → Cloud computing; • **Hardware** → Error detection and error correction;

Additional Key Words and Phrases: Reproducibility, scientific computing, computational science, workflow, workflows, scientific workflows, scientific workflow, replicability, reproducible

ACM Reference format:
Peter Ivie and Douglas Thain. 2018. Reproducibility in Scientific Computing. *ACM Comput. Surv.* 51, 3, Article 63 (July 2018), 36 pages.
https://doi.org/10.1145/3186266

1 INTRODUCTION

Reproducibility is a central requirement of the scientific process. To validate the claims made in a publication, enough information must be provided that a skeptic can perform an equivalent experiment and thereby confirm or refute the results.

In recent years, serious questions have been raised about the reproducibility of scientific work in general, and scientific computing more specifically. While there appears to be a general sense that

Authors' addresses: P. Ivie and D. Thain, Computer Science and Engineering, University of Notre Dame, Notre Dame, IN 46556.

https://doi.org/10.1145/3186266

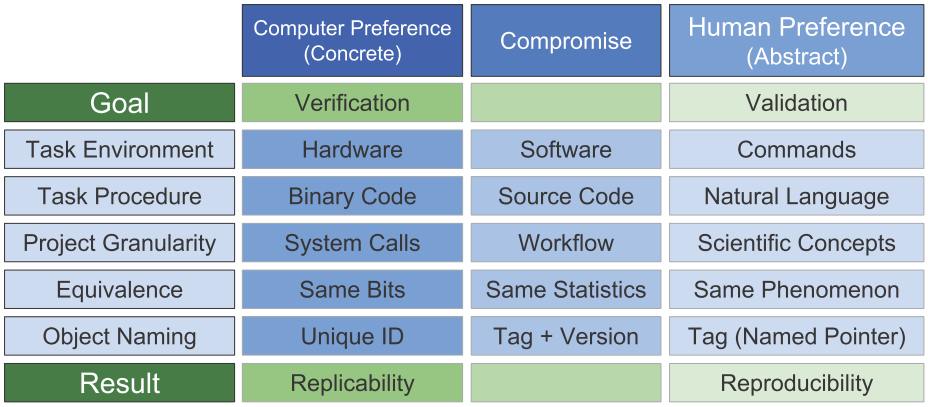

Fig. 1. Overall perspective for this survey.

most scientific computing is not as easily reproduced as it could be, there is no general agreement on what, precisely, reproducibility entails, and what mechanisms are needed to achieve it.

In principle, computational experiments should be easy to reproduce, when compared with physical experiments. Assuming that a computer is a deterministic machine, then simply applying the same program to the same inputs on an equivalent architecture should yield equivalent results. Many design principles and recommendations surrounding scientific computing have been encouraged [19] for years. But in practice, the complexity of today's software and hardware makes it surprisingly difficult to even accurately describe the inputs, construct a deterministic program, or identify equivalent hardware. Many of the difficulties stem from a need to simultaneously satisfy the needs of both the computer and a human as summarized in Figure 1, rather than being able to focus exclusively on one or the other.

In this article, we survey the state of reproducibility in scientific computing, which we define as computing applied to the physical sciences (biology, chemistry, physics, and so on) for the purposes of simulation or data analysis. We seek to illuminate the following fundamental questions:

—What constitutes reproducibility in scientific computing?
—What are the technical barriers to replicating a single command or a complex workflow?
—What is the significance of the names for reproducibility?
—What techniques are available to achieve reproducibility?
—What are open problems in the realm of reproducibility?

To limit the scope of this article, we focus on technical problems in scientific computing, and refer the reader to other publications that address the broader questions of publication habits [147], the role of funding agencies [107, 152], fraud [37, 103], legal issues [153], and similar questions [127]. Elements of scientific discovery that do not involve computers are not the focus of this article.

Computer aided engineering (CAE) is a similar field where computer software is used to help perform engineering analysis tasks, rather than scientific workflows. While not applied to the same domain, some of the concepts used can have applicability to scientific computing. For example, executing simulations of a model can be used to evaluate the validity of the model in both CAE [105] and scientific computing [129].

2 PERSPECTIVES ON REPRODUCIBILITY

It is commonly expressed that reproducibility of computational science is a desirable quality, and that there is a need to move beyond the printed paper as a means of communicating results [133, 151].

> "An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures" [25].

This type of information can get very complex and detailed very quickly, but from a user perspective it can be easier than it sounds with the proper tools and mindset.

> It is a big chore for one researcher to reproduce the analysis and computational results of another [...] I discovered that this problem has a simple technological solution: illustrations (figures) in a technical document are made by programs and command scripts that along with required data should be linked to the document itself [...] This is hardly any extra work for the author, but it makes the document much more valuable to readers who possess the document in electronic form because they are able to track down the computations that lead to the illustrations. [34]

Some argue that this situation has reached crisis proportions:

In the biotech industry, Amgen [14] attempted to confirm the findings in 53 "landmark" articles in cancer research. These attempts were not merely computational, but also involved working in the original labs under the direction of the original authors in attempts to resolve discrepant findings. They only succeeded with 10% of them. In pharmaceuticals, Bayer [137] had slightly better results and were able to verify 21% of 67 different projects. These efforts involved scientific research where the computational resources required were generally low. For computation-based scientific research, part of this problem should be easier, since it is trivial to make many copies of a piece of software once it has been written. However, for many reasons, this is not necessarily the case.

More recently, researchers at the University of Arizona [138] considered the repeatability of 402 ACM papers published in computer systems conferences. In this article, minimal repeatability was defined simply as the ability to download and build the source code within a reasonable amount of time. They were able to build 32.3% of them within 30 minutes. 15.9% more took over 30 minutes and 5.7% more with additional but reasonable effort. The code failed to build in 2.2% of the cases, and the authors declined to provide code in 7.5% of the cases. 36.3% of the authors never responded to requests for the code. The subjects of the study were invited to post corrections or addenda to the material, and the responses resulted in a wide variety of strong opinions about the procedure.

All these numbers are definitely dismal, but is it really a crisis? Future computational science is likely to become even more complex [142] and resource intensive, making reproducibility even more challenging. To understand why this is such a problem, we will explore some of the reasons for, and benefits that come from making research reproducible.

2.1 How is Reproducibility Defined?

A wide variety of authors have defined reproducibility and related terms in somewhat different ways. [1, 48, 55, 72, 73, 92, 116, 129, 154, 169, 173]. Although complete consensus has not been achieved on these terms, we will use them in the following way:

To *replicate* an experiment [48] is to carry out exactly the same task as the original researcher, with the expectation that the result will be the same. In scientific computing, exact replication

would constitute building the same program with the same compiler running on the same hardware and the same operating system as the original. Obviously, it may be difficult or impossible to replicate every last detail. Seemingly innocuous details (like the system time [91]) may affect the final result.

To *reproduce* an experiment [173] is to carry out tasks that are equivalent in substance to the original, but may differ in ways that are not expected to be significant to the final result. In scientific computing, these differences could range from minor to sweeping. One attempt to reproduce might run the same version of the software on a new version of an operating system, while another attempt to reproduce might involve writing a new piece of software that implements the same algorithm.

The terms *verify* and *validate* are often used interchangeably. In the broader literature [17, 129], they are used to indicate technical correctness and fitness for purpose, respectively. In the context of scientific computing reproducibility, we define *verification* as the task of replicating an experiment to see if it produces the claimed output, while *validation* is the task of evaluating a result to see if the author's conclusions are warranted. One experiment *corroborates* another when they reach the same overall conclusions.

When the components underlying an experiment are easily named and shared, it becomes possible to make use of them in other contexts. This is known as *variation* or *reuse* or *extension*.

The term *provenance* is used broadly to describe, retrospectively, the many potential sources of input or variation to a program. For example, when a program is run in a distributed system, it may be desirable to record the incidental details of the machine on which it ran (architecture, operating system, system time) in case those details are later found to be significant. Or, if a program B consumes input data X that was the output of a previous program A, then it may be fruitful to record that $A \rightarrow X \rightarrow B$ to note that the output of B originally depended on the output of A. [26, 53, 69, 98, 104, 143, 148, 156].

A *deterministic* program always produces the same result when run with the same input in the same computing environment. Some programs are non-deterministic by design: for example, a Monte Carlo simulation uses a random number generator to evaluate a function with randomly chosen inputs. Other programs are non-deterministic by accident: concurrency, operating system services, or the vagaries of floating point math may all introduce differences where they are not desired [52]. For example, even a Monte Carlo simulation with a fixed seed can produce non-deterministic results [91]. There may be ways to avoid some sources of non-determinism, but it is difficult (if not impossible) to avoid all of them. Efforts can be made to detect [13], or mitigate such behavior, even at large scales [29], but there is still a need to support better reproducibility at the system level.

2.1.1 Equivalence. We must also be careful to define what constitutes the "same result" when comparing two experiments (see Figure 2):

- Two experiments could produce the *exact same bits*.
- Two experiments could produce the *same data* in the sense that they encode the same numeric contents, but differ in some irrelevant detail. For example, an output file might incidentally contain the system time and the name of the user who ran the program.
- Two experiments could produce *statistically equivalent results*, in that the numeric values are different, but they both conform to the same statistical distribution, modulo some error tolerance.
- Two experiments could observe the *same phenomenon* but not the same data.

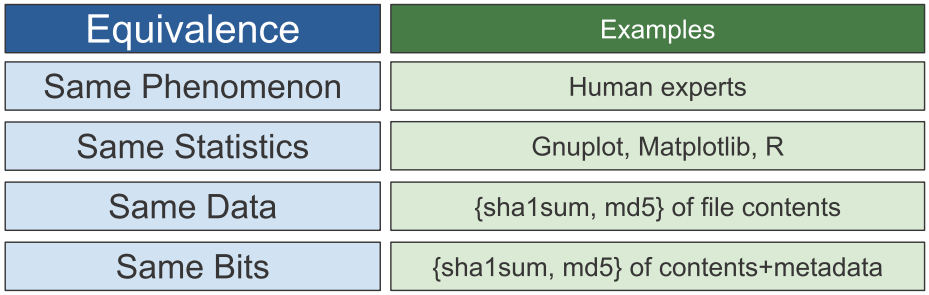

Fig. 2. Equivalence can be gauged with different methods depending on the desired focus. Discussed in detail in Section 2.1.1.

These distinctions have an important bearing on whether a result can be verified automatically. If equivalence is defined by the same bits or the same data, then simple technical tools can be used to perform the comparisons. Evaluating statistically equivalent results requires a domain-specific tool, while comparing phenomena requires a human with domain-specific knowledge. Consequently, it is desirable to achieve reproducibility at the lowest level at which it is feasible, so that verification can be performed automatically.

A part of reproducibility is communicating methods and intent [48], in addition to communicating how to obtain identical results. When tools for logical equivalence are not available, the burden of comparison rests on the scientist.

When the computer itself is the object of study, then performance or resource consumption may be the primary result. In other cases, issues of performance are relevant in terms of cost and/or convenience, but are not the focus of their research. Various systems exist that are both appropriate for computational science and have a focus on maximizing, measuring, or repeating performance goals. [28, 47, 93, 95, 102, 150] *Repeatability* can refer to the ability to get the same performance [1] in the presence of changing conditions in the underlying system. We will assume that for the general case, a focus more on reproducibility has benefits that outweigh the advantages of a focus more on performance.

Topics discussed in other works, but not addressed in the article include best practices beyond the technical aspects [154] and roles of not only the scientist, but the funding agency and the journal editor [173].

Several authors [116, 134] have presented these reproducibility concepts as a spectrum starting with replicability by a single researcher as the minimum level of scientific integrity, and increasing through verification, reproduction, validation, extension, and reuse by many researchers. Each stage requires a greater amount of work but has increasing value to the community at large.

2.2 Why Should Computing Be Reproducible?

There are a variety of reasons underlying the need for reproducibility:

To verify (or disprove) other's results. Mesirov [122] argues that the basic function of a paper is to both announce some result and convince the reader that the result is correct. However, Ioannidis [89] claims that the majority of published research findings are false, due to small sample sizes, statistical noise, confirmation bias, and publication bias. In most fields, peer-reviews serve to evaluate whether the work, as described, is sound, significant, and interesting. With rare exceptions, reviewers do not have the time, inclination, or skills to perform and verify the work described in a paper, particularly if it requires access to unusual or expensive methods and facilities.

However, scientific computing has the unique advantage that any computational activity is potentially reproducible, given the same code and input data and execution on a compatible machine.

This has given rise to the concept of "reproducible research" [33] or an "executable paper" [20, 27] in which the source code and data used to reach a conclusion are coupled and distributed with the paper itself. In principle, this should allow the reviewer and the reader to carry out the same action and evaluate the conclusions.

This concept has been offered as part of a number of special issues and efforts, but has not been accepted broadly by research communities as of this writing. This may be due to the fact that peer review considers more broadly the novelty, significance, and correctness of a work. For example, Leek [106] notes that merely re-running the same code does not guarantee that the research results are correct. There are also many cases where accessibility of the code and data is not sufficient: the results may require access to specialized or high performance hardware, and may still require a large amount of time or other resources to complete.

To verify one's own results. In practice, we have encountered relatively few researchers who wish to actively develop an adversarial relationship with others by disproving their work. However, some have argued that a healthy distrust of one's own work should drive reproducibility:

> We do not take even our own observations quite seriously, or accept them as scientific observations, until we have repeated and tested them. Only by such repetitions can we convince ourselves that we are not dealing with a mere isolated coincidence, but with events which, on account of their regularity and reproducibility, are in principle intersubjectively testable. [136]

To improve one's own productivity. Some researchers perceive that efforts to make their research reproducible will result in decreased productivity [10] as effort is shifted towards technologies instead of their primary work. Others have argued the opposite. For example, Jon Claerbout (who coined the term "reproducible research") made the following statement after many years working towards that end:

> It takes some effort to organize your research to be reproducible. We found that although the effort seems to be directed to helping other people stand up on your shoulders, the principal beneficiary is generally the author herself. This is because time turns each one of us into another person, and by making effort to communicate with strangers, we help ourselves to communicate with our future selves. [32]

To enable extension by others. Frequently, one researcher may wish to build upon another's work positively by augmenting it or evaluating it against a new dataset or situation. This is easier said than done. Even when two researchers working contemporaneously can share notes and advice, moving a code from one institution to another can take months before the same setup is "working" in the new context [70]. It becomes even harder when the original researcher is no longer available: they may have graduated, have taken a new job, or have died. If we desire to enable reproducibility on the scale of 10–20 years [25], significant care is needed to record all the necessary details. A focus on the human side of scientific computing [81] can also make it so that others can understand and incorporate published research into their future efforts.

To survive technology evolution. Many research codes depend on a large number of subcomponents like libraries, compilers, and runtime systems that must be independently installed, configured, and tested on a given operating system. A large computing site must occasionally go through an upgrade cycle to activate new hardware, change system facilities, or upgrade the operating system. These are frequently not backwards-compatible changes, and so all the supporting

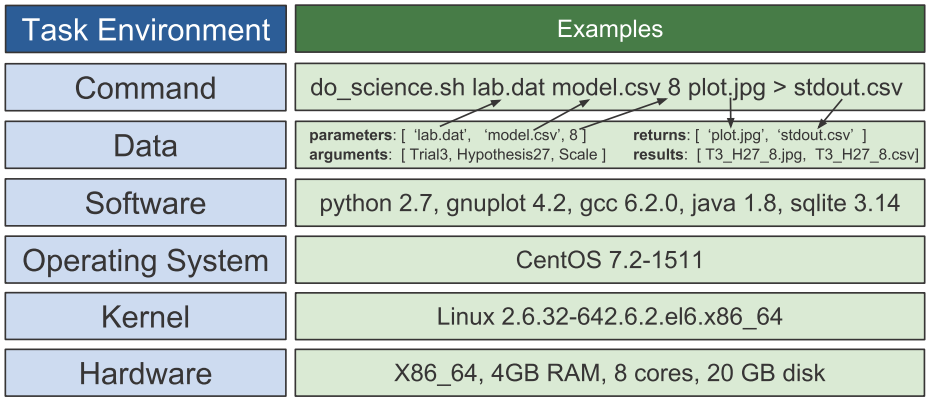

Fig. 3. Examples of environmental components at various levels. More details in Sections 3.1.1–3.1.6.

components must be re-built to accommodate the new environment. The unsuspecting user may face an enormous amount of work to reconstruct all these components. Reproducibility techniques can assist in recreating the dependency tree (and testing it) after a major upgrade.

To enable community maintenance and support. A code developed by a single researcher typically has a short productive lifetime. Keeping the code working on multiple platforms and relevant to current research trends takes time, and eventually the researcher moves on to other activities, leaving "orphan" code behind. However, if reproducibility techniques make it easy to execute a code in many different contexts, responsibility for the code can be held by a larger community. When multiple stakeholders are familiar with the code, technical problems are more easily solved. Even automated techniques can be employed to perform maintenance on an experiment when the research is adequately reproducible [57]. By publishing reproducible research, ownership of maintenance is effectively transferred to the community [59] level. This allows the publishing scientist(s) to focus more on future work than previous work.

In summary, computational scientists are often encouraged to make their research reproducible so that that other scientists can *verify, reproduce,* and *extend* their computational experiments, but there may be personal benefits also.

3 TECHNICAL BARRIERS TO REPLICATING A SINGLE COMMAND

Let us begin by considering the technical challenges of reproducing just a single command. Suppose that an end user connects to a university computing facility and enters the following command:

```
do_science.sh lab.dat model.csv 8 plot.jpg > stdout.csv
```

From the user's perspective, the command string is the only visible evidence of the program. The command itself provides the most superficial form of replicability: by entering the exact same command into the same terminal later that day, there is a good chance that exactly the same outputs will be produced.

However, there is no guarantee that the same command applied by a different user on the same machine, much less a different user on a different machine, will succeed at all, much less produce the same output. This is because the command string replies on a large number of dependencies in the form of hardware, software, and data, as shown in Figure 3. Some of these dependencies

are explicitly mentioned on the command line (like the file `model.csv`) while others are implicitly provided by the system.

The following sections cover such environmental challenges, in addition to challenges connected with abstractions, run-time anomalies, verification of results, and a discussion about whether source or binary code should be the target of preservation.

3.1 Environment

We define the *environment* as both the system resources and the domain methods used to perform the computational side of scientific research. The scope at which systems preserve or describe the environment varies widely. Research is more likely to be reproducible when all levels of the environment are preserved or at least identified.

3.1.1 Command Scope. A "do_science.sh" script can contain all information needed to replicate the experiment. However, this approach can mask valuable information from the user, making it difficult for another scientist to extend the research to explore or build on the experiment. Requiring additional parameters that are handled by the script may seem to overcomplicate an experiment, but doing so communicates those decisions made by the original researcher that are deemed most relevant. *Parameterization* can be an important tool for extension. If crafted carefully, parameters can give both the original researcher and others the ability to easily explore the parameter space to gain confidence in the validity of the research. Figure 3 starts with an example of a parameterized command at the top. Parameters can be numbers or strings in the command scope, but in the data scope (the next section) the parameters can refer to files.

3.1.2 Data Scope. Most scientific research involves some kind of input or starting data in addition to the final generated results. For reproducibility purposes, we will mostly consider this data to be in the form of files, but it could come in the form of Unix standard output, literal parameters, and so on.

This data could involve network dependencies that can make reproducibility more challenging. For replicability, these dependencies might be satisfied by recording the data retrieved over the network and then simulating the network in subsequent runs of the experiment. However, for other changing factors (especially to the domain methods and original data), this approach becomes less likely to capture the network resource adequately. It might be necessary to capture and transfer the entire database from behind the network resource to ensure that the workflow will still be reproducible in the presence of changes.

Authorization issues are another common challenge with data, either from a security or privacy perspective. Authorization keys are sometimes kept in pre-determined file or location, and certain data files may include private information. Such information should normally be excluded from a publication, but without it, the research is not reproducible. Accepting this information as an input parameter can identify a need for the information without publishing the sensitive data, enhancing reproducibility. This could be done in the form of a template.

Templates are programs that expect input parameters for dynamically specifying the data the program is supposed to operate on. They can be helpful for the original researcher when the input data is updated incrementally or to compare different datasets. This form of parameterization is also useful, for extensible reproducibility, when other researchers have different original data and want to use that to evaluate the domain methods with.

Using a single command with lots of parameters can get confusing for large experiments. In the interest of extensible reproducibility, and to simplify things for the original researcher, it is advisable to break down the experiment into smaller parts and organize them in a *workflow* [41, 109, 165]. Different levels at which a workflow can be composed are discussed in Section 4.0.1.

Descriptions that involve both the command and data scopes are similar to *functions* (consider Figure 3), where an external name needs to be given to all *arguments* (for the inputs) and the *results* (for the outputs). The internal names used for that data inside the function are considered *parameters* (for the inputs) and *returns* (for the outputs). This separation between internal and external names can be important for workflows because sometimes legacy software expects input from fixed filenames and uses fixed locations for generated files. In these cases, and when a template is used multiple times as part of a workflow, the naming of data files can become complicated from a workflow perspective. More on the issue of naming can be found in Section 5.

3.1.3 Software Scope. Part of the challenge with software dependencies is that different versions of a particular software program or library are mostly compatible, but always include some changes. Even newer versions of software that claim to be backwards compatible may have unintended differences that can affect reproducibility. Software names can be ambiguously used without version numbers for ease of use by the scientists, but for reproducibility, all necessary software should be uniquely identified to ensure consistent behavior (more in Section 5). *Packages* can be used to make this process easier and can include any combination of elements from the command, data, and software scopes, as shown in Figure 3.

Also, scientists generally prefer to focus on their science and care less about lower level resource management [79] handled by system administrators. They are more interested in the destination than the journey, in part due to the exploratory nature of scientific research. Scientific research is more often marked by a desire for *infrastructure independence* than with an emphasis on *benchmarks* and how quickly a system can execute a workflow. Readers interested in infrastructure and performance can find more information about system-centric workflow systems and distributed test beds in Reference [24].

If a given research experiment is infrastructure independent, it will be more easily reproducible. However, scientists sometimes decide they need a little more control over the system resources. At this point, the boundary between system administrator and domain scientist comes into question, which in turn makes the responsibility for reproducibility more ambiguous. What software should be provided by system administrators versus how much control should scientists have in setting up their own domain specific software on generic computing resources?

From a reproducibility perspective, the scientist is generally unaware of modifications made by the system administrators. Even the system administrator may not put a priority on tracking all aspects of the system's configuration. The line between domain methods and system resources also varies between domains, making it difficult to come up with a reproducibility solution appropriate for all domains.

3.1.4 Operating System Scope. Occasionally, the scientist will want a different version of the operating system than is available on provided resources, but this is often beyond their control. *Containers* such as Docker [120], Kubernetes [23], and Mesos [82] have emerged to address the need for users to have easier access to specific versions of operating systems. Container popularity is evidence that there is a need for this level of specificity in a description of how computational science is performed.

Containers also depend on a specific kernel to work. This means that by specifying an appropriate container for computation research, the kernel is also specified. But it also means that a container depending on one kernel cannot be initiated on a computer running a different kernel.

3.1.5 Kernel Scope. Virtual machine images can satisfy the need for all system software in addition to providing virtual support for some hardware requirements of computational research. However, the overhead of instantiating virtual machines can be prohibitively high, especially for

short running tasks if a virtual machine is instantiated for each task. This can cause domain scientists to gear the workflow toward performance, with larger tasks, rather than extensible reproducibility, with logically sized tasks. These larger tasks are likely to be more obscure to other scientists than those designed with logical domain science granularity (see Section 4.0.1).

3.1.6 Hardware Scope. For workflows executed on a single machine with no network dependencies, the computer itself could be preserved in a museum or library as a part of reproducibility [126], but this is clearly not feasible especially for large datasets analyzed on distributed systems.

Another aspect of the hardware scope not normally handled at the other scopes, is the concept of finite resources. A scientist focused on domain specific issues can neglect to preserve the memory, cpu, disk, and perhaps network resources needed for a workflow and its parts. This information is also difficult for system administrators to track because additional resources are required to monitor the resources used by a workflow. Both groups are disincentivized to preserve information this detailed, but it may be difficult for another scientist to reproduce the research without this information.

3.2 Source Code or Binary Code?

When attempting to preserve a workflow for reproducibility, decisions must be made about when to preserve source code and when to preserve the binary code that was generated by compiling the source code.

Considering reproducibility, source code seems like the obvious choice. But the compiler then becomes an important part of the workflow and measures need to be taken to ensure that the compiler is available and can execute on future compute resources. Sometime in the future, it might be easier to find a modern replacement for a compiler than it would be to find a modern way to execute the the binary code. At some point in this recursive problem, assumptions may need to be made about what will be available in the future. So, just preserving the compiler doesn't completely ensure reproducibility in the long term.

Source code more easily communicates to colleagues what each task is doing and lends itself more easily to modification by those other scientists, making it a better choice for reproducibility. In fact, important information about the science is embedded in the source code, whether through comments, structure, or naming. This information is ignored by the computer as irrelevant, but could be considered a collection of facts that help support the claims made in the published research. Those facts can add knowledge about a workflow in general and also software components individually, especially when the components are novel in the scientific domain.

To use the source code, it must be converted to binary code by a compiler, which can take a significant amount of time. If the compilation is done on each compute node in a distributed system, the compile time can add significant costs to executing the workflow. This makes preserving the binary code a better choice if replicability is all that is needed, not reproducibility.

However, there is no need for a rule that states one option must be chosen at the expense of the other. If both forms are preserved, the preferred option can be chosen later on, when the needs are more clear. For data-intensive workflows, preserving the source code, compiler, and the binary code is unlikely to make an unreasonable addition to the total storage or communication costs.

Having both options also makes it more likely that one or the other will provide the level of replicability/reproducibility needed at some future date on some future compute resources. Indeed, having more than two levels of abstraction where the highest level describes the task or workflow in very broad terms might allow for a task or workflow to be re-created in a situation when neither the source code nor the binary code can be executed.

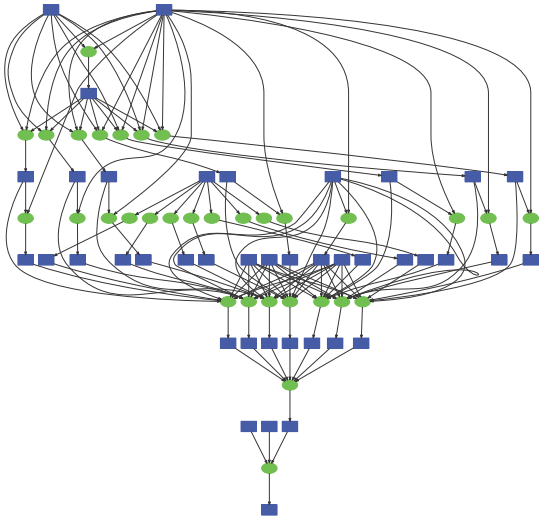

Fig. 4. Visualization of a Genome Analysis Workflow. Data or information (indicated by squares) and actions or processes (indicated by circles) are connected by a derivation tree, where the data at the root is source data, and any subset of the remaining generated data components could be considered the results.

4 TECHNICAL BARRIERS TO REPRODUCING A WORKFLOW

The Workflow Management Coalition [84] defines a workflow as the computerized facilitation or automation of a business process, in whole or part. For a scientific workflow, the business process is an experiment with a focus on scientific discovery, innovation, and/or invention. A set of procedural rules describe the processing and generation of documents or information. We also focus specifically on solutions for scientific workflows which are data-intensive [148]. While there may be some scientific computing efforts that would not be considered workflows, there is value [149] in applying workflow concepts wherever computers are used as a part of the scientific research process.

A visualization of a workflow for Genome Analysis is shown in Figure 4. Data or information (indicated by squares) and actions or processes (indicated by circles) are connected by a derivation tree, where the data at the root is source data, and any subset of the remaining generated data components can be considered the results.

"The workflow programming paradigm is seen as a means of managing the complexity in defining the analysis, executing the necessary computations on distributed resources, collecting information about the analysis results, and providing means to record and reproduce the scientific analysis" [158].

A workflow management system [165] provides a bridge between the work people do and the work the computers do. They are important [115] in making computational science more convenient for scientists, but they can also help improve reproducibility. Unfortunately, there are many workflow management systems, and there is no common or accepted format or procedure by which a workflow should be recorded or shared.

4.0.1 Project Granularity. The size of each step in a scientific workflow can be as small as a system call, or as large as a single command that performs a complex system of hidden computations.

Project Granularity	Operations	Examples
System Calls	sys_open, sys_gettimeofday	Traced by ReproZip, CDE
Middleware Operations	split/merge, map/reduce	Available in Kepler, Triana, Taverna
Domain Tasks	simulate(x), analyze(y)	Supported by Makeflow, Pegasus
Workflow	BWA-GATK, Monte Carlo	Shared through github.com, Galaxy
Workflow History	Get Checkpoint D	Archived in Prune

Fig. 5. Computations can and are preserved at various granularities. Each option has certain benefits and shortcomings (seeSections 4.0.1–4.0.5).

For extensible reproducibility, the granularity should be domain specific and chosen by a scientist to reflect the granularity of the scientific concepts involved. Many systems impose restrictions on the granularity, making it more difficult to use the workflow as a way to communicate the details of the research between scientists. However, those restrictions can also make it easier to use and more effective for a specific class of user [38]. Each of the levels of granularity shown in Figure 5 have advantages, but also disadvantages, which can be a barrier to reproducibility. In addition, the existence of so many options can be a barrier, as a scientist accustomed to using one level may have difficult adapting to another.

4.0.2 Granularity: System Calls. One simple solution for preserving a workflow is to trace [30, 135] and log all system calls (such as **sys_open, sys_stat, sys_gettimeofday, sys_getuid**, and so on) during the execution of a workflow. This system call log can be used to identify which files were actually used for the workflow. The remaining files can be excluded from a package or image that contains the environment the workflow is to be executed in.

While this approach can provide replication for deterministic workflows, it may not work if the workflow is modified or if part of the workflow is non-deterministic, since different library files might be needed in a subsequent execution.

Even after eliminating excess files for a given package, duplicates will exist across packages that are only slightly different from each. This becomes a storage problem as a workflow evolves through progressive iterations.

Sharing a workflow at this level can definitely provide replicability, but it is difficult for a colleague to understand what the workflow does. The log itself can be valuable for a very experienced user, but for a domain scientist, it is probably only useful as a last resort when other more coarsely organized workflow descriptions fail.

4.0.3 Granularity: Middleware Operations. Another solution is to allow a middleware designer to choose which logical operations can be applied to data. More complex operations must be created by the scientist composing new operations using a combination of provided logical operations, such as *merge/split* operations, or *map/reduce* operations.

Kepler [4] is an extensible system for the design and execution of scientific workflows with a focus on GUI presentation. *Directors* are execution models with plug-ins that manage *actors* or tasks (sources, sinks, transformers, analytical steps, compute steps). The Triana [157] workflow environment is designed for managing distributed applications (P2P, Grid, middleware toolkits). It works at a web services level (GUI for connecting tasks), but more complex services can be built on the ones provided by the system. Taverna [88] is a tool for building and running workflows that is also based on web services.

While better than working with system calls, these low-level operations are typically less abstract than a domain scientist would prefer to deal with, and make it difficult to understand the science without an abstraction at higher levels.

4.0.4 Granularity: Domain Tasks. Alternatively, a scientist can choose what happens in a task. As an example, one task could be designed to simulate events, while another analyzes them. Parameters on the task could include things like the number of events, their type, and a seed. Using abstractions at a domain level [40] makes it easier for other users to understand the workflow when it is shared with them. This flexibility makes domain tasks a good granularity for extensible reproducibility.

Problem Solving Environments (PSEs) provide all the computational facilities necessary to solve a target class of problem. Users can use the language of the target class of problems, while the PSE fills in the details with appropriate hardware and software. The user does not need to have specialized knowledge of the underlying hardware or software. [66, 67, 97]

In effect, they separate problems and solutions from the hardware and software that carries out the solution. This helps the user focus on their domain [19], and could also support the use of advancements in hardware and software without additional effort from the user.

Such systems are particularly well suited for education [125, 160] because they allow the focus to be on concepts, not programming. PSEs can also be used in distributed computing, allowing a researcher to access more computing power with less effort [65].

CAE [2] is a similar approach applied specifically to engineering, and can have some applicability to scientific computing.

4.0.5 Granularity: Workflow and History. A scientist can group tasks together into a specific workflow. This level of abstraction can identify the results that are used for a publication, but is more effective for reproducibility when it includes components at lower levels of granularity. There is also significant information to be found in the evolution of workflows (see Section 4.0.8). Comparing workflows after small changes or comparing between researchers, can identify portions that are similar [90] or identical. This enhances reproducibility by allowing scientists to focus on differences rather than similar portions.

However, a scientist can't keep everything forever. The archiving of knowledge is generally done by libraries who keep records through books. Their influence is expanding into the digital realm, but their exact role is still unsure. We generally assume that once a decision has been made to keep something, it is easy to keep it forever. However, libraries constantly have to make decisions about what to keep and what to discard, and deal with such issues as copyright and access. Another publication [141] provides more information about such issues and how they apply to research data.

4.0.6 Abstract versus Concrete. The chosen method for describing a workflow can fall along a spectrum between abstract and concrete, or can incorporate both abstract and concrete elements that are connected together. Abstractions can allow the scientist to work with high-level concepts that can be later compiled into more concrete components [114]. The more abstract the workflow description is, the easier it is for scientists, making it more likely to be extensibly reproducible. High-level visualizations can be used as a guiding tool for solving specific problems [18].

But at the same time, an abstraction can leave room for unexpected behavior, putting even replicability at risk. For example, a graphical user interface can be bad for reproducibility because unexpected behaviors may go unnoticed.

At the same time, a more abstract workflow might be chosen because it can be more adaptive to changes in the runtime environment. Especially with the exploratory nature of scientific

computing, the exact number of computations needed in a particular step may be unknown in advance. In such situations, the importance of verifying results becomes even more significant.

4.0.7 Data Management. Dataset sizes are increasing in all fields involving computational science—maybe not on the order of petabytes such as with high energy physics, but usually large enough to merit distributed systems for processing and sometimes even simply storage. Most version control systems for managing source code are designed to fit on a single machine. In addition, with a focus on managing lines of code, data is typically treated as an inscrutable blob of bytes.

A workflow is of no use without the data it depends on, but with big data, the data must often be kept separate from information on the workflow with such version control systems. The connection between the two can be the first component to break down when attempting to reproduce the workflow. With a little bit of personification [74], some have come to accept that data needs more public attention.

In addition to large input datasets, the data generated by the workflow might be too large to share with others practically or efficiently. Without the final datasets, it is difficult for a collaborator to verify the results of an attempt to reproduce a workflow.

Data provenance refers to the derivation history of a data product starting from its original sources. Derivation steps could include database queries, command line strings, executable files, or other similar actions eventually producing some data result. Specific examples of ways to preserve provenance for computational tasks are available [63]. But the exact metadata recorded varies widely, depending both on the requirements of the system and the purpose for the provenance. In fact, a full survey [148] is dedicated to this topic. Data provenance does not always include sufficient detail to re-execute the history of operations.

4.0.8 Evolution. Designing a scientific workflow is an evolutionary process. Recording the evolution can be valuable in communicating the validity of research. If another scientist wants to try changing some parameter in the workflow, the evolution history might reveal that the path has already been tried. In addition, seeing the various workflow attempts can help to convince other scientists that sufficient attention has been given to the parameter space surrounding the final research. The absence of this data is a missed opportunity for more extensible reproducibility.

This evolutionary workflow data could also be useful in bi-directional research sharing. If multiple research groups are working in a similar vein, they can benefit from each others' efforts. However, this type of data could easily grow beyond the scientist's capacity to preserve the workflow evolution data without attention on the minimum data that must be recorded, so that derived data does not have to be stored indefinitely.

4.1 Workflow Execution

Certain methods for executing the workflow can introduce problems with both reproducibility and in the validity of the scientific research itself. Research is vulnerable to *measurement bias* in many different forms [124], especially when the scientist can execute code manually.

Automating the execution of all parts of the workflow can resolve some of the measurement bias, while at the same time saving time and being more convenient for the researcher. However, any model used to automate a workflow can be restrictive to the researcher. This could be because there is significant performance overhead or the automation language is prohibitively complex for the scientist. Whatever the reason, scientists can feel a great deal of temptation to execute at least some of their workflow manually, making it hard to maintain a reproducible representation of the workflow.

Even an automated workflow can run into isolation issues where data is available on the original computing resources, but is not available in the workflow description. For example, either the

scientist or system administrator may be unaware of the dependency on some resource. In this case, more isolation between the workflow and the user space would help with reproducibility because such problems would have to be resolved before the experiment could complete execution in the first place.

In other cases, the resource could also be intentionally unavailable based on proprietary or privacy restrictions in place. In this situation, the isolation between the workflow and the user space can actually get in the way of reproducibility.

Also, the size of a resource could make it impractical for inclusion in the workflow, or for performance reasons, the resource could have been made available on a site specific resource such as a shared or distributed file system. In such cases, the resource should be identified in the workflow to satisfy isolation, but the actual run-time connection to the resource may need to be more flexible. Finding a balance for isolation that is appropriate for reproducibility is difficult.

There are different ways [175] to make sure data is getting to the right places for execution. In a *user-directed* approach, users must identify file locations in the specification and a method for obtaining them, if not already available. In a *centralized* approach, a central repository holds all the data and each execution node must transfer files to/from there. In a *mediated* approach, a central repository holds only meta data about files and their locations. The files themselves can be distributed across many nodes, reducing the bottleneck on the central repository. In a *peer-to-peer* approach, no central repository exists, so each execution node has a list of neighbors that can be queried for the data.

The centralized approach is the most reproducible, assuming the central repository remains available. The peer-to-peer approach does not require a central repository, but a workflow might not be reproducible if even one of the input files can't be found. The mediated approach is also more risky since the node(s) containing the actual data could be unavailable even if the central repository is working. After long periods of time (such as decades), a user-directed approach might need to be a fallback for reproducibility purposes, in the event that resources needed for the other methods are no longer available.

Alternatively, the code can be moved to the data when the data is big and the code is small. This can be a big boon to performance in some cases, but might not be supported by a workflow management system designed for a data movement approach. It is also possible that this mode of execution is not available for another scientist using different resources, so relying on code movement would decrease the chances of a workflow being reproducible.

Fault tolerance is a phrase often used when describing distributed systems that can handle the failure of some nodes in the system. But various levels of fault tolerance can be appropriate, depending on the scientific domain. For example, in some stages of high energy physics workflows, not all tasks need to be completed to consider that stage complete. This type of behavior can be a challenge for reproducibility because the failed tasks might vary between scientists. It can be difficult to determine whether a workflow is indeed reproducible when different individual failures exist in two workflow executions, but the number of failures is acceptable for both executions.

If multiple users are a part of a single workflow, there can be a great deal of confusion when someone updates the workflow at an earlier stage. A scientist working on the later portions of the workflow will need to somehow merge that update into what they are working on. The update may or may not affect the final results of the workflow, but it can be complex and challenging to figure out an appropriate and convenient time at which to incorporate the update.

5 NAMING

Due to the exploratory nature of scientific discovery, attempting to introduce too much organization into the workflow too early can be wasteful. Scientists may prefer to wait on coming up

with a name for certain components (such as "analysis" or "simulation") until they are sure their value in the workflow is proven. Or they might only give human-centric names to the most significant components. In the meantime, the computer still needs to distinguish between the remaining objects and must most likely also group or link them together.

In addition, the names used to identify certain objects at one point in time can be repurposed to refer to new objects as the workflow evolves. A workflow could also evolve in two different directions (perhaps by multiple scientists, or by one scientist trying two different ideas). Given two workflows with similar origins, it is difficult to identify which components are the same, and which are different. If this comparison becomes too difficult, the effort needed to incorporate a colleague's research can be greater than the benefits that may come. In fact, a bad experience attempting such a comparison can lead to a perception that no benefits from a colleague's research can outweigh that cost.

5.1 Namespaces

Each scientist, working individually, has full control over their own namespace, which allows them to ensure that names identify the appropriate entity or entities. The home directory for a particular user on a computer is an example of a namespace. A user generally has complete control over the file and folder names in their home directory (except for a few reserved names and characters such as the "." and ".." folders and certain characters not permitted in file names). Users can also create new namespaces when they create sub-directories.

An entity named "analysis' by one scientist could refer to a completely different entity by another scientist. When their namespaces are separate, this is not an issue, but when sharing research methods with other scientists, these collisions can be impossible for a computer to resolve automatically.

One solution for the merging of namespaces is to take an approach similar to how directories' hierarchies work. Individual scientists are allowed to continue with their own namespace, and a parent organization (or a super-namespace) is created to distinguish between the individual namespaces when more than one namespace is involved. An identifier for each namespace (such as a path or folder name) can be prefixed before the rest of the name.

However, the prefix can become tedious for frequently used entities that happen to have been created in a different namespace. A scientist may want to give a new name for that entity in their own namespace, which can also become confusing because now a single entity has multiple names. If colleagues communicate with names that are only appropriate in their own namespace, there can be a great of confusion about what is being referred to.

5.2 Persistent Identifiers

A scientist will occasionally need to move workflow files to new storage locations. If the naming of workflow components is tied too closely to their pathname, this hinders the ability to compare different evolutions of a workflow. On the internet, the ability to move and replace lower-level network components is ensured by creating a hierarchy of names, each level of which can be modified without affecting the higher-level names. Such higher-level names (such as DOIs) are especially relevant when reproducibility is taken into consideration.

Each device on a network is an entity, identified at the lowest level, by a *MAC address* (Media Access Control), which is essentially a name for an entity on a network (see bottom of Figure 6). The MAC address is a 6-byte number, which is typically displayed to the user in hexadecimal notation (for example, 48:65:6c:6c:6f:21). The purpose of the MAC address is to uniquely identify that specific network device worldwide.

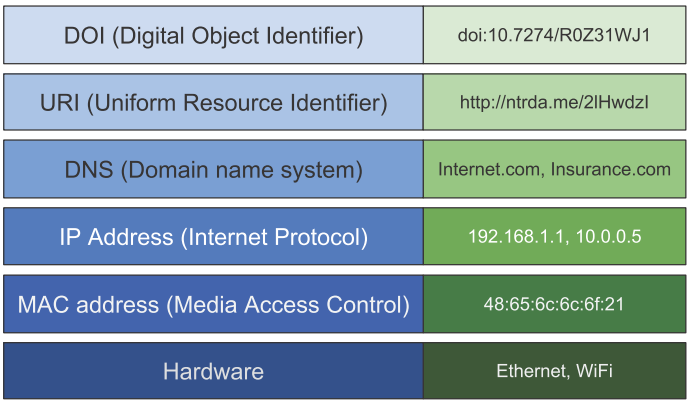

Fig. 6. Examples of network and internet entity naming.

If that device fails and is replaced, the new name must be propagated to everyone who used the old name. *IP addresses* were created to mitigate this problem. They are a new higher-level entity or abstraction designed to identify a computer (rather than a device), and can be mapped to a specific device as needed behind the scenes.

In general, lower-level entity names are typically more effective for replicability. They help to identify the exact set of operations needed to re-execute an experiment precisely. The MAC address is better for doing exactly what was done before, but an IP address makes it easier to account for later changes to the system. This makes IP addresses a better choice when extensible reproducibility is the goal, since the MAC addresses will not be appropriate on another set of computing resources.

But IP addresses are still difficult for people to remember, so domain names were created. *Domain names* are more human-friendly names that identify a new entity called a site (rather than a computer), and can be mapped to IP addresses.

Each of these namespaces is managed by an organization which allocates sub-namespaces to other organizations. For example the first three bytes of a MAC address identify the organization (such as a hardware manufacturer), which manages the last three bytes. And with domain names, top-level domains (such as .com and .org) have control over the namespaces below that top level.

The identifiers for these namespaces are directly tied to a location, a domain name resolving down to an individual network device at any given point in time. But there is also a need for persistent identifiers that are separated from the location of the entity they refer to. And in addition to identifiers for physical entities, such as a network device, there is also a need for identifiers (as names) for any type of digital entity.

A URI (Uniform Resource Identifier) is designed to identify any resources, with a URL being the most common type of URI. A URL is connected to a location through the domain name embedded in its identifier, but a URI does not have to include a location. The Handle System [155] is also part of the URI specification and is a namespace for global persistent and unique identifiers with a specific data type, but no changeable attributes such as location, ownership, permissions, or timestamps. One implementation of the Handle System is the DOI (Digital Object Identifier) system. A DOI can be an identifier for any type of entity, and is associated with a URL, but the URL can change as needed.

A similar hierarchical approach could enhance extensible reproducibility by providing names that persist across scientific domains, but such a system is not yet in place. DOIs are often used for this purpose (to provide a global name for some entity) and are a great step towards reproducibility. However, their ability to change can also be a problem over time, because there is no guarantee that a given DOI will hold the same contents in the future as it did at the time a scientific paper was published.

5.3 Immutable Identifiers

The ability to change the entity that a name refers to can make it difficult, from a historical perspective, to effectively communicate what entities were used to generate research results. For a given name, if an entity is replaced after or even while results are generated, referring to that name later on will likely prevent reproducibility if the system does not prevent this behavior.

In between a persistent identifier (which is designed for user convenience) and the location of an entity (which is all the computer needs) is an immutable identifier that allows multiple copies of the entity to exist in various locations, but prevents revisions or alterations to the entity that could prevent reproducibility.

If a central authority exists that manages unique entities, the authority can assign an immutable identifier to each entity using methods ranging from an incrementing number to UUIDs. In a distributed environment, a checksum of only the significant parts of an entity can provide an immutable identifier using an agreed upon algorithm. But a significant amount of forethought is needed since there are many checksum algorithms available, and it can be difficult to distinguish between significant and changeable attributes of an entity.

5.4 Overloading

Choosing an appropriate system for generating identifiers is further complicated by the challenges that come from having too many names for an entity or too many entities for a name.

Reproducibility can be more expensive when there is more than one appropriate name for an entity. In workflow descriptions, this *entity overloading* can cause extra network traffic, storage space, and computational resources because the computer (or a person) might treat them as separate entities. These problems usually cause inefficiencies, but have no effect on reproducibility in the presence of sufficient time and resources. File systems that support file linking can deal with this problem by allowing a single file entity to appear to be in multiple folders and/or have multiple filenames.

On the other end of this spectrum, when namespaces are not clearly defined or managed *name overloading* can occur. This is a much more significant reproducibility problem than entity overloading. Imagine a folder on your computer with a single filename that points to two file entities. When attempting to access the filename, the computer is unable to decide which actual file to open. Perhaps the file system could prompt the user to choose one over the other, but if accessed in a script, this is not possible.

The actual solution in most file systems is that the new file entity replaces the old one. The user is often prompted to make sure this is the desired behavior, but once replaced, the old file entity with that name is no longer available. Even if the file system keeps all versions [145], it is difficult to resolve filenames when the version is ambiguous or without a desired timestamp, and the system is forced to make a guess.

This is a common occurrence in evolving workflows when there is a progression of the specific details (or entities) used to achieve some generic purpose. From the scientist's perspective, each progressive version is an improvement on the last, so the most recent one should be used. However, even a small change in a single entity can drastically change the final results. So, for reproducibility,

it is important to uniquely identify entities in the workflow so the correct entities get used for a historical workflow. This means that a name alone may not be sufficient if previous versions need to still be available.

5.5 Versioning

A example of this problem that is a common source of reproducibility problems is the evolving versions of software libraries. The intent may be for all versions to be backwards compatible, but there can still be subtle changes that alter results. In addition, for the purposes of reproducibility, forward compatibility might be needed if the original scientist used a newer version of the library than a later scientist. A version hierarchy (ex. 1.2.10) is often used to distinguish between updates that are more or less likely to break a system that relies on the library. To identify relevant entities, this version "number" should be used in connection with the name of the library to uniquely identify an entity such that the name and version is an immutable snapshot of that library at a specific point in time. The version hierarchy has meaning to the user, but a computer typically sees it as no different than what could be achieved with a timestamp or a number that auto-increments with each new version. The combination of a name and a version provides both; an appropriate name for the user to understand the purpose of an entity, and an immutable entity that should be used by the computer.

5.6 Hashing

Another problem is that computers are unable to detect similarities between entities the same way humans can. However, they are very good at quickly identifying when two objects are in fact the same identical entity. A hash function can be used to map data of some arbitrary size to an identifier (or hash key) of some fixed size. To be useful, hash functions must always produce the same fixed identifier for a given data entity. When using a hash function to name a data entity, there is almost no possibility for entity overloading even without a central authority. A perfect hash function will always produce a unique identifier with no collisions (i.e., no name overloading). While a universally perfect hash function is unlikely, a wide range of hashing algorithms are available with various collision probabilities, many of which are appropriate for almost all situations.

Unfortunately, despite the advantages, humans find it difficult to work with hash keys because they appear to be random and need to be fairly large to provide sufficient assurance that collisions will be avoided.

5.7 Distributed Hashing

To avoid a central authority, a hash function can be agreed upon as a way to uniquely identify relevant entities. For example, git [110] uses hashes of file and directory content to generate a hash that uniquely identifies each commit. A description of the commit is highly encouraged, but is not intended to be unique or to serve as a key for finding the commit. The description is solely for the benefit of the user.

A central authority is avoided because all participating computers agree upon a programmatic division of entities. Each computer maintains the data agreed upon by the group's hashing function, and the system relies on that function to ensure entities can be found using appropriate names.

5.8 Tags

Tags aren't much different than names, but the word is used to describe objects with no attempt at or expectation of uniqueness. In fact, they are used more to group objects than distinguish

between them. In a way, they are the opposite of versioning. Versions are a computer-friendly addition to human-friendly names, while tags are a human- friendly addition to computer-friendly unique identifiers (using hashes or an authority). Tags are also typically only relevant within some localized namespace.

5.9 Lineage

Another approach to naming that is particularly attractive for reproducible research is to embed the lineage or history of a particular entity in the name itself. For example, in a Merkle Tree [121], the leaves are ordinary data entities without names, and every other node has a hash key as its name. The hash key is generated using the hash of the sum of the hashes from the child nodes. Other technologies that incorporate similar techniques include Git [110] and CVMFS [16].

Attempts have been made to create a universal identifier [70] for computational results. But there are still many different approaches being taken and it seems like there is more divergence in methods occurring than there is convergence. The conflicting goals of naming versus entity identification make it difficult for both humans and computers to find and distinguish between computational entities.

It should be clear at this point that there are many barriers to overcome, each of which can distract a scientist from their domain of expertise. All combined together, the barriers seem to form a wall that effectively prevents most computational science from being reproducible.

6 TECHNIQUES FOR ACHIEVING REPRODUCIBILITY

Scientists should carefully choose a reproducibility technique that aligns with their goals for replicability and/or reproducibility. Some techniques are convenient and provide replicability, but are too complex to be effective for reproducibility. Others offer limited flexibility, but provide a high level of reproducibility. The following reproducibility techniques draw from related ideals in computer science or have a focus on satisfying domain specific needs.

6.1 Track All Operations

One simple solution for preserving a workflow is to trace [30, 86, 135] and log all system calls during the execution of a workflow. As mentioned before, the system call log can be used to reduce the size of a package or image, enabling the execution environment. This approach works for replicability, but is not always resilient enough for extensible reproducibility. However, there are other considerations that can make this approach desirable.

Transparent Result Caching [164] is one way to automatically track the dependencies that are explicitly stated in a Makefile, so the user can just use ordinary shell scripts to execute workflows. If the method needed to generate results is recorded, the results themselves can be treated as a cache. When an input changes, cascading results can be prompted for or automatically regenerated. Certain dangers exist with this approach, such as when there are non-deterministic operations, network communications, and/or failed tasks.

Nectar [77] is a more modern system designed for data centers. It can detect duplicate execution requests before they get sent to execution nodes and instead return the cached results. Since VisTrails [12] is designed to generate images interactively, caching the results can be a large benefit so that all historically explored images don't have to be stored indefinitely. Logothetis [111] also addresses saving and reusing previous computations to reduce the amount of traffic that has to be aggregated. They can also use that information to detect effective file equivalence even when a checksum comparison says the files are different.

6.2 Track All Actions within a Walled Garden

Environment dependencies can also be resolved by requiring all execution [51] to occur on a shared, public testbed. In such a system, a workflow can be tracked at a very high level. For example, an interactive text editor [100] could track lines of a script as they are entered by the user and execute them in the background on behalf of the user.

To prevent conflicts between user and system control of the environment, such executions are more reproducible if performed in a clean sandbox. In other words, execution should occur in an environment that is both separate from user space and loaded without implicit dependencies.

This type of system can even support multiple users [71] with the possibility of some shared state between them. This virtually ensures reproducibility and extension of workflows in the short term. However, flexibility is very limited and scalability is often out of the hands of the researchers, and eventually the shared system will need to be updated, requiring new consideration of all archived workflows.

6.3 Track All Actions from An Achievable Initial State

Some methods assume an implied initial state, but it is a good idea to make the initial state more explicit. Recording changes in a way to support undo [176] can provide a way to achieve a consistent initial state that can be shared with others. If the ability to replay the changes is included, then there may also be the ability to revise the replay instructions to support extensions to research rather than just replication. This replay ability also provides good efficiency during execution with the ability to look at a particular operation in more detail later [42], without having to store all details on the fly.

6.4 Execute a Detailed Specification

Rather than let the user perform or request operations on the fly, the user can be expected to get organized and plan their workflow in advance, evolving the workflow as needed to handle the exploratory nature of computational research. Such a plan should start with a clear recipe for a repeatable environment, and then domain appropriately sized building blocks can be used to advance the state of the workflow using that environment.

The relationship between all tasks in a workflow and their dependencies can often be fully described using a DAG (directed acyclic graph). Some tasks may need to be run in *series* (a specific order) if they depend on the results of other tasks to be executed. These series tasks contribute to the height of a DAG that describes the workflow. Tasks without such dependencies on other components can be run in *parallel*, and contribute to the width of a DAG.

Some workflow systems [3] only support tasks that can fit into a DAG structure. Such systems can optimize the use of resources during execution because all required tasks are known in advance. The DAG could be extended to allow for additional functionality, such as with conditional DAGs [128], but adding additional components to a DAG may reduce or negate optimizations that a DAG based system can do.

Tasks whose execution is *conditional* or tasks within *loops* (see Figure 7) are beyond the scope of a DAG. However, a higher-level abstraction can be used to describe conditional or looped tasks without losing DAG optimizations as long as the conditions and loops can be decided before execution begins. For example, 2 tasks inside a loop that iterates exactly 10 times, where one of the tasks only executes on odd iterations, can be easily translated into 15 tasks in a DAG, even if each iteration of the loop depends on the results of the previous iteration.

However, any loop or condition that relies on runtime results cannot be directly translated into a DAG before execution. Such workflows must be handled on a more on-demand basis. Prediction

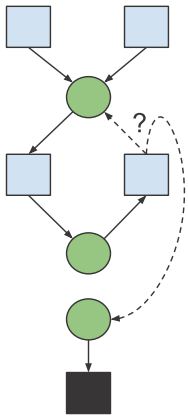

Fig. 7. Directional graph with conditional loop.

or estimation may be possible [113, 117], but the full benefits of a DAG are not available. If other elements beyond a DAG are needed, the description must be more abstract.

A language-based workflow description can support large and/or complex workflows that can be directly shared with collaborators. Also, a person with programming experience may be able to easily write a program (perhaps in their preferred scripting language) that generates the desired workflow description in the target language. This allows a user to create more complex (and more abstract) workflows than are available in the workflow language itself.

Some systems created and use custom languages designed specifically for their workflow system [64, 68, 130, 140]. Others focus more on adapting the workflow into a standard language (XML for example) [5, 8, 21, 171, 174]. Somewhere in the middle there are standardized languages created for use in multiple workflow systems [6, 167]. A language can even be interactive with the ability to run complex workflows programmatically, while at the same time preserving the workflow in various convenient languages even before the code is executed [139].

There is great value in designing the language to approximate the scientist's natural language [85]. This is more convenient for the original scientist, but is also easier for collaborators to understand. Taken to the extreme, such systems [97] might remove the need for scientists to deal with any kind of programming, since the language provides all scientific needs, and the low-level details are handled by individuals outside the scientific domain.

However, users often prefer a graph-based system (rather than language based) for workflows (especially new users) because learning a new language can be difficult. A graph-based approach [80] also typically abstracts lower-level details away from the user, forcing/allowing them to focus on higher-level concepts that are more applicable to their scientific research.

Graph-based modeling is sometimes implemented in the form of Petri nets [83, 166]. FlowManager [9] is one example. More recently, UML (Unified Modeling Language) [11, 49, 144] has also been used to address some of the issues with Petri nets.

The best of both graph- and language-based approaches can be available by exposing both options to the user. In addition, allowing multiple representations of the workflow [114] can offer new insights and capabilities. Grid-Flow [76] includes a Petri-net based interface and a programmable Grid-Flow Description Language. XRL/Flower [170] also uses Petri-nets, but uses XML to support standard parsing and validation of the language.

For large/complex workflows, a graph-based approach can become unwieldy due to the vast details available. To help graphically modeled systems support larger and more complex workflows, low-level details [131] should be abstracted away from the user. A system of templates [83] can be used to specify sub-portions of a workflow in a hierarchy of abstractions. Triana [157] supplies a graphical user interface that allows users to drag and drop tools and connect them to inputs and outputs. Tools can then be "grouped" together to both simplify the visualization of the workflow and to support an abstraction hierarchy. Kepler [113] supports "abstract components," which collapse the details of a subworkflow to tame complexity. By using WSFL (web services flow language) to compose workflow elements, each composition can be used hierarchically as a web service for higher-level compositions [108]. Another interesting approach [69] is to attempt to automatically generate higher-level abstractions on top of the low-level provenance.

Ideally, a user could describe a workflow in very abstract terms and some domain-specific system could flesh out the details automatically. Pegasus [40] combined with Chimera [61] support this type of approach. Given a somewhat abstract description of inputs and the desired outputs, Chimera can automatically look up operations in a database and those operations can then be used to accomplish the desired behavior. Pegasus then marshals resources to execute the workflow, and generate the desired results.

For all of the techniques, but especially for a detailed specification, it is helpful to ensure that a history of the evolution of the workflow is somehow recorded. In software development, this is done with a version control system where changes to the software are periodically recorded. This is so that the state at important points in time can be obtained even after future changes have been made. Such checkpoints can be automatic (so the user doesn't forget), or the user must develop a habit of choosing appropriate times to record the state. More frequent checkpoints make it easier to isolate and undo bad changes, but they are more work to create and sort through. Websites such as http://github.com and http://bitbucket.org have made it convenient to share this information with collaborators, and are often used for scientific collaboration. However, such systems can break down when large amounts of data are involved as they are designed more for source code than for data.

Even with reproducible research there is no guarantee that the research results will be correct [106].

6.5 Verify and/or Validate the Final State

The above-mentioned Pegasus+Chimera method of workflow abstraction is also an example of another broad technique for reproducibility. This technique is to pay very close attention to the final result and less to the steps along the way.

Take for example, the difference between Puppet [112] and Chef [159]. Both are systems widely used for system configuration. Chef takes an imperative if-then approach where the system administrator designs a sequence of statements that specify exactly what actions should be performed to get the computer in the desired state. On the other hand, Puppet allows the system administrator to declaratively say what packages should be on a computer, and Puppet has some freedom in determining the best way to install those packages.

A declarative evaluation of scientific results is usually merited, with or without an imperative list of steps required to get there. The responsibility to ensure that scientific research results pass declarative needs generally rests completely on scientists. With tools that can assist or automate some of this process, scientists can be responsible to ensure those tools are applicable and appropriate. Sometimes verification and validation are as simple as comparing newly generated results to results which have already been verified or validated.

If it is possible to automatically determine that results from a replication attempt are equivalent to the results from the original research, then the exact methods used to achieve those results may need less scrutiny. Verifying and/or validating the final state could be the only technique needed to reproduce very simple workflows, but for large workflows, this technique would be more effective when used in conjunction with another technique focused on the steps along the way.

Ideally, a scientific workflow will be fully deterministic and the generated results will always be bitwise identical. However, in practice, there are many sources of non-determinism [91] that contribute to results being different after a workflow replication attempt. The degree to which small sources of non-determinism affect an entire experiment [35, 44, 98] is important to consider.

Some tools help with a comparison, but still require a user to make a final decision on equivalence. For example, a comparison tool called *sfvplotdiff* [57] is used in the Madagascar project to compare a plot generated by an established version of a workflow with a plot generated by an extended evolution of that workflow. Special care is given to tolerate precision differences [52] between the computers executing the workflow, so that a scientist can observe only significant differences between the plots. Then the scientist can more easily decide whether the differences are justified depending on how the workflow was extended.

6.6 Require Formal Dependencies

Relying on conventions is more convenient (for most users) than creating elaborately configured frameworks. However, preferring configuration over convention is helpful for reproducibility.

At the programming language level, dependencies on libraries can be specified using commands like import, include, and require. However, it is unlikely that the programming language will locate and retrieve those dependencies if they have not already been installed. A container image [120] or virtual machine image can be coupled with the code to satisfy those dependencies.

The dependencies can also be specified in a functional manner [46] if the dependencies need to be more granular than a single image. Alternatively, a set of commonly used dependencies can be bundled together [119] so that a single reference to the bundle can indirectly include all of the dependencies from the hardware to the command level.

A system that can embed a full environment specification into each component of a workflow [43, 90] enables users to ensure that all required dependencies are included. If the system only executes the workflow using the fully specified environments, then any generated results have a high likelihood of being at least replicable. In addition to providing the ability to replicate component execution on similar hardware, this form of encapsulation [85] could include subcomponents such as a compiler, to make the environment specification more portable to hardware systems that are less similar to the original ones used. This also allows for more separation between domain science and computer science [97], reducing the need for domain scientists to be involved in programming in favor of focusing on their domain, and also increasing the reproducibility.

6.7 Validate Continuously

In software engineering, continuous integration [62] is when developers flag working updates to a shared codebase as soon as they are ready, and then several times a day, updates from all developers are merged, tested, and put into production. Various websites [31, 94, 161] can be used to support continuous integration. Comprehensive testing (or validation) is typically automated so that unexpected problems can be quickly identified before going live.

This concept can be applied (in part) to scientific workflows for collaboration. While the merging is less likely to be valuable several times a day (compared to internet applications), the measures taken to make that possible can simplify the collaboration process enough to make it more effective for reproducibility. For this to work, a few practices need to be adopted.

There is a big difference between a scientist manually reporting the command they executed and the scientist requesting a command to be executed with the system automatically reporting the command used. When the scientist manually reports a command, there is always a possibility that something was left out, or that something was changed after the report was made. When the reporting is automatic, there may still be room for implicit dependencies built into the system, but those can be easier to track and resolve than transient changes based on what a user types into the command line.

Automation is a good first step to ensure that a workflow, and any sub-components, are correctly reported in connection with some results [39]. By some definitions [84], automation is the purpose of creating a workflow in the first place.

One of the earliest and most prevalent ways to automate a workflow is the Unix *make* [118] utility. Early attempts [146] to encourage reproducible research using *make* resulted in some success, but over 38% of the files (figures) were not reproducible. In addition, the system was fairly complex, including scripts in various languages and LaTeX macros, the combination of which made it difficult for other scientists to reproduce [60].

More recent attempts [58] using the more modern automation utility *SCons* [101] (based on Python) were eventually more successful [57], but that success isn't necessarily tied to those utilities alone. There has been more pressure to make research reproducible in recent years, and some of the successes could be attributed to that momentum.

In many cases, the benefits of automation may actually contribute to the overall "ease of experimenting" [24] while at the same time making great strides toward reproducibility.

In software engineering, test-driven development is where the programmer creates a test for desired behavior before the behavior is even implemented [54]. This ensures that even if changes are made to the software, the desired behavior is preserved. Scientific research is generally too exploratory to be able to define the desired workflow fully in advance. However, once some research has been published reproducibly, it can be treated as a test in the sense that it can be used as a basis for comparison as other researchers attempt to replicate or extend the research. Tests are vital for performing continuous integration [50].

Various systems [7, 75, 99, 123] designed for automatic deployment and task execution could be used to assist in efforts to validate continuously.

The Madagascar project [57] applies this concept to reproducible research by running tests whenever someone submits a modification to preserved research. If automatic validation fails and no one in the community steps in to correct the error, the project is removed from the set of maintained research workflows.

Some workflow systems are specifically geared toward automating and preserving the generation of graphs and figures to be shown in a publication. One approach combines Git And Org-Mode [150] to handle the execution of a workflow that is fully documented as it is created. The final result is similar to a lab notebook and can be published with all details on how to reproduce it. Similarly, Paper Mâché [20] manages a workflow and a LaTeX or .doc file, directly inserting images generated by the workflow into the document for publication. Vistrails [27] is designed to interactively generate images so that the viewer can explore visualization with custom arguments to available parameters. These are tools that provide some automation, but final verification/validation is performed by a user.

6.8 Make the Environment Explicit

The computing environment used by most scientists is provided by system administrators who attempt to balance the needs of many users when making decisions about how to provision hardware. When scientists need system resources that are not already provided, they may ask the

system administrator to install or procure those new resources for them. They might go as far as choosing a specific version of an operating system or even bleeding edge hardware for their research. If those resources are not automatically provided, some scientists will even take on some of the role of system administrator to obtain those resources themselves. Virtual machines and containers are relatively new technologies that give scientists more flexibility. Some of these new technologies improve reproducibility at the same time.

6.8.1 Hardware Provisioning. The traditional way to provide compute resources to scientists is to consistently provision dedicated hardware for each research group. This can include a detailed list of imperative operations [159] that describe how to install the environment directly on hardware. Alternatively, a declarative specification [112, 119] lists all the components that are needed in the environment, but does not dictate how they are to be installed.

It is also important to consider the community, reliability, and usability [132] of hardware provisioning specifications, as they can get very complicated.

6.8.2 Virtual Machines. A more recent technique involves provisioning the hardware with a generic system that allows virtual machines to dynamically be instantiated as needed. This provides the scientist the most flexibility. But more importantly, it automatically provides an easy to preserve copy of the full environment used in the original workflow execution. This virtual machine image can then be easily shared [87] with colleagues using cloud services.

This approach is quite effective when the research can reasonable fit in a single virtual machine. However, a networked system can break down with large datasets or with research that requires distributed execution to complete in a reasonable timeframe. Vagrant [78] goes one step further by including complex network configurations in addition to managing software within the virtual environment. However, there is a high likelihood that many unneeded files will get included in the virtual machine image, making the image excessively large and more difficult to curate and share.

6.8.3 Containers. A container image [120] serves many of the same purposes as a virtual machine image. But while the virtual machine image is a single file with everything needed, a container image is a progression of new packages added to previous ones. As such, a container can have less overhead (especially when booting) because it may be able to rely on already running parts of an operating system. Except for the fixed operating system kernel, a container can provide what appears to be a completely independent system. This allows more efficient use of RAM and faster startup times, in addition to layered file systems and common files that make disk usage more efficient.

6.8.4 Package Management. A package management system mostly handles changes to installed software libraries. However, most package management systems require root access, which means that access to such systems must be restricted [162] for security reasons. This contributes to a barrier between system administrators and scientists leading to confusion when attempting to identify and share the environment.

However a few non-root package managers [36, 46] are emerging that take a functional approach to setting up appropriate environments. Giving the scientist such control can help ensure that upgrades don't sneak in leading to unexpected results, and can help even with heterogeneous [168] devices.

6.8.5 Functions. Organizing a workflow into functions is one way to parameterize tasks into abstract components that perform domain-specific operations and are organized in a way logical to scientists. A simple command can be treated as a function if the executable is designed to work that way (see Figures 3 and 8).

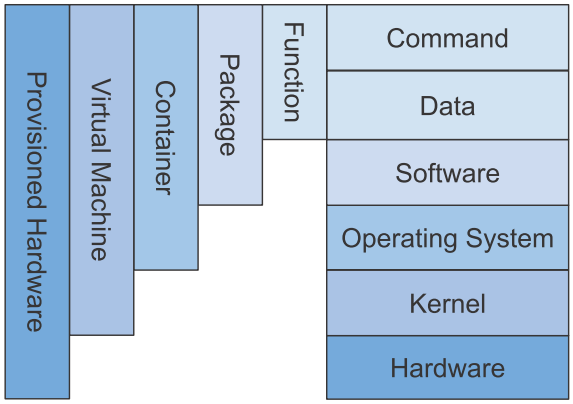

Fig. 8. Environment scopes. *The environment can describe anything from a command involving some data, to the full computational stack down to the hardware specifications.*

6.8.6 Distributed Systems. For workflows that don't fit in a single node, a connecting structure needs to exist between instances of an environment. To execute complex multi-cloud and multi-VM applications reproducibly, cloudinit.d [22] can launch, configure, monitor, and repair a set of interdependent virtual machines over multiple IaaS clouds. A launch plan describes a series of run levels, each of which contain tasks that can be run in parallel. A service handles the launching, configuring, and status of VMs, starting with package management tools like Yum or configuration management tools like Puppet. In the case where failure is detected and repair actions are needed, cloudinit.d only restarts the affected sections of a launch plan.

Whatever the method for defining and creating environments, doing so not only helps with reproducibility, but also allows the scientist to delegate the responsibility of providing reproducible system resources.

7 OPEN PROBLEMS FOR REPRODUCIBILITY

Despite advances in the tools currently available to enable reproducibility, many authors can't even replicate their own results after a year [45], let alone enable others reproduce them 10–20 years from now. There is little external motivation to ensure scientific computing publications are reproducible. Perhaps a metric needs to be created to measure reproducibility so that a sizable prize [15] could be offered to the most reproducible scientific computing publication. In the absence of a funding source, maybe publishers could simply start offering a Most Reproducible Paper Award similar to the Best Paper Awards commonly given. A little notoriety could go a long way in encouraging scientists to strive for reproducibility. Efforts could also be made to improve scientific computing tools so that reproducibility was less difficult and ambiguous.

7.1 Automatic Verification of Results

The ability to automatically compare two separate executions of a workflow is more valuable than it may seem at first. Such comparisons, however, should not be limited to a simple conclusion of identical or not. Enabling multiple levels of equivalence at each level of the workflow can allow the scientist to be informed as to whether or not changes to system resources, timing, or other incidental variances have affected the results in a meaningful way.

If the comparison involves many executions (more than just two), the scientist might be able to make conclusions about whether or not a given level of equivalence is normal at that stage of the workflow, or indicates a new variation that should be examined more closely. In addition, automating comparisons at each level would make it trivial to identify specific components that are causing variations, whereas a manual comparison at the end of the complete workflow would necessitate reviewing all stages to identify an issue.

Automating the comparisons could also introduce the ability to establish scientific controls. Iterative executions while one parameter changes could then identify not only parameters, but parameter values that have an affect on reproducibility. Automated verification could also enable continuous integration, which in turn could give the scientist earlier notification that some change had a significant effect on the results. This could both save time for the scientist and reduce the cost of compute resources.

As threats to reproducibility are identified more precisely, there is an opportunity to enhance reproducibility by making adjustments to domain methods. This could even lead to concrete metrics on how reproducible a workflow is, similar to code coverage metrics that help encourage software developers to have tests for as much of their code as possible.

7.2 Performance, Scalability and Efficiency

With automatic verification of results, a focus on performance, scalability, and efficiency can all be undertaken without a concern that such optimizations will hamper the reproducibility. This may not be the primary concern of most scientists today, but current trends indicate that big data needs will only increase over time. A workflow that is sufficient to handle current databases may be unable to perform the same operations on larger databases that become available, making it hard to extend research to future input data. However, sometimes the most scalable approach for the future is at odds with the most efficient approach today. This balance will be a constant struggle, but with automatic verification, some of the struggle can at least be automated.

7.3 Infrastructure Independence

It is hard to even imagine any kind of workflow description that is entirely infrastructure independent. In addition, infrastructure independence is typically at odds with HPC (High Performance Computing). HPC often focuses on using the commands that provide the best performance for the specific infrastructure being used.

However, to increase the chances that a system will handle future infrastructures that are yet to be invented, some level of infrastructure independence is important. One way to achieve this is to make the architecture and/or workflow tasks generic enough that they can be adapted as needed. Some sort of middleware could observe both sides and make appropriate conversions to ensure proper execution.

These conversions could be especially dangerous to the reproducibility of a workflow, so it would be important to couple efforts in this area with automated verification. However, this combination could make it possible to optimize performance without sacrificing reproducibility.

7.4 Curation of Workflow Evolution

Even research that was fully reproducible at the time of publication may cease to be usable in the ensuing year as a result of unexpected hardware or software evolution. The Madagascar project [59] and observations of its use after a couple of years [57] make a strong arguments for making research preservation a community effort, rather than placing the burden entirely on the original researcher.

This is not an easy task and generic open source software techniques [56] are not always applicable. Strides have been made and lessons learned in very specific situations [163], but more needs to be done for scientific workflows as a whole.

7.5 Usability

There are clearly usable solutions available for many of the reproducibility problems that have been discussed. And it could be argued that improvements to usability are always possible. However, usability does not seem to be a high priority in many of the the tools created and used in computational science. More focus on usability would clearly help to increase the adoption of tools that are designed to reproducibly execute workflows and can satisfy the other needs of computational scientists.

8 CONCLUSIONS

For a workflow that can run on a single machine, solutions exist for replicability, but for distributed systems, and/or for reproducibility, existing systems are generally inadequate for scientists who are experts in their scientific domain and not experts on computer systems.

Tracking system calls is one approach for replicability that requires little effort from the scientist, but for large-scale, complex workflows, many issues can arise if the work needs to be distributed, especially if the distributed resources are heterogenous. For such situations, a more flexible approach is needed to utilize such distributed and evolving resources. Within tasks, both source code and binary code are important to communicate both scientific intent and concrete implementation, respectively. Task sizes can vary from a single system call (which is easy for a computer to deal with), to everything behind a single high-level scientific concept with some parameters (for user convenience). For most systems, a hierarchy of tasks interacting with each other is likely to be appropriate. These interactions can be described as a workflow with a language or graphical-based approach, the former being more expressive, and the latter being more convenient, but both can be effective in communicating the real science to collaborators.

Each element of the workflow should have names that are sufficiently abstract and flexible for the user, but at the same time sufficiently unique for a computer. This could include name and version pairs for the user and computer, respectively, or unique identifiers with human readable elements embedded in them.

Many existing efforts [85] to provide frameworks, middleware, and environments to support computational science are available. With some effort, scientists can organize their research to support parameterization so that the research might be extended in new directions. But in addition to such efforts, which would allow new scientists to explore new avenues, there also needs to be an effort to communicate intent and higher-level purpose to those new scientists. This might be done in publications, or built into the workflow, but these methods definitely require additional work from the scientist. However, this effort might not be wasted, as sometimes an author can be communicating with their future selves when they clarify their workflow.

With that in mind and with a focus on usability, additional tools are needed to reduce the intellectual load on scientists so that they and their collaborators can focus on their scientific domain instead of on the computer science. In general, the less work the scientist has to do to execute their research workflows and evolutions of their workflows, the more likely it is that their collaborators will be able to accept and benefit from that research. This includes any efforts towards validation, infrastructure independent, and performant execution, recording, sharing, and synchronizing of workflows.

Beyond technological improvements supporting reproducibility, a shift in intent and a greater focus on reproducibility needs to be adopted by scientists. Reproducible research needs to be

perceived by all involved as a more effective contribution to science rather than an inconvenient and unachievable ideal. Scientists could also benefit from more exposure to any good software development practices [172] that are not included in their education or training. There is no guarantee that even the most reproducible techniques available today will be reproducible at any given date in the future. Therefore, a community of experts is needed who are willing to maintain a collection of relevant research. This community would secure funds, decide what research is no longer relevant, what research needs to be updated to accommodate technological advances, and develop additional tools to encourage new commitments to the reproducibility of computational scientific research.

REFERENCES

[1] Ali Abedi, Andrew Heard, and Tim Brecht. 2015. Conducting repeatable experiments and fair comparisons using 802.11 n MIMO networks. *ACM SIGOPS Oper. Syst. Rev.* 49, 1 (2015), 41–50.

[2] Erika Abraham, Hadas Kress-Gazit, Lorenzo Natale, and Armando Tacchella. 2017. Computer-assisted engineering for robotics and autonomous systems (dagstuhl seminar 17071). In *Dagstuhl Reports*, Vol. 7. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[3] Michael Albrecht, Patrick Donnelly, Peter Bui, and Douglas Thain. 2012. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the Workshop on Scalable Workflow Enactment Engines and Technologies (SWEET'12) at ACM SIGMOD*.

[4] Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludascher, and Steve Mock. 2004. Kepler: An extensible system for design and execution of scientific workflows. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*. IEEE, 423–424.

[5] Kaizar Amin, Gregor Von Laszewski, Mihael Hategan, Nestor J. Zaluzec, Shawn Hampton, and Albert Rossi. 2004. Gridant: A client-controllable grid workflow system. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences, 2004.* IEEE, 1–10.

[6] Peter Amstutz, Michael R. Crusoe, Nebojša Tijanić, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, Matt Scales, Stian Soiland-Reyes, and Luka Stojanovic. 2016. *Common Workflow Language* (version 1.0). (July 2016). DOI: http://dx.doi.org/10.6084/m9.figshare.3115156.v2

[7] Paul Anderson and Edmund Smith. 2005. Configuration tools: Working together. In *LISA*. 31–37.

[8] Matjaz B. Juric, Benny Mathew, and Poornachandra G. Sarang. 2006. Business process execution language for web services: an architect and developer's guide to orchestrating web services using BPEL4WS. Packt Publishing Ltd.

[9] Lerina Aversano, Aniello Cimitile, Pierpaolo Gallucci, and Maria Luisa Villani. 2002. FlowManager: A workflow management system based on petri nets. In *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC'02)*. IEEE, 1054–1059.

[10] Lorena A. Barba. 2016. The hard road to reproducibility. *Science* 354, 6308 (2016), 142–142.

[11] Ricardo Melo Bastos and Duncan Dubugras A. Ruiz. 2002. Extending UML activity diagram for workflow modeling in production systems. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)*. IEEE, 3786–3795.

[12] Louis Bavoil, Steven P. Callahan, Patricia J. Crossno, Juliana Freire, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. 2005. Vistrails: Enabling interactive multiple-view visualizations. In *IEEE Vis. 2005*. IEEE, 135–142.

[13] Olivier Beaumont, Jocelyne Erhel, and Bernard Philippe. 2000. Aquarels: A problem-solving environment for validating scientific software. In *Enabling Technologies for Computational Science*. Springer, 351–362.

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

[15] Robert Bell, Jim Bennett, Yehuda Koren, and Chris Volinsky. 2009. The million dollar programming prize. *IEEE Spectrum* 46, 5 (2009), 28–33.

[16] Jakob Blomer, Predrag Buncic, and Thomas Fuhrmann. 2011. CernVM-FS: Delivering scientific software to globally distributed computing resources. In *Proceedings of the 1st International Workshop on Network-Aware Data Management*. ACM, 49–56.

[17] Barry Boehm. 1989. Software risk management. In *Proceedings of the European Software Engineering Conference*. Springer, 1–19.

[18] Choompol BOONMEE and Shigeo KAWATA. 1998. Computer-assisted simulation environment for partial-differential-equation problem. *Trans. Japan Soc. Comput. Eng. Sci.* (1998), 19980002–19980002.

[19] Randall Bramley, Bruce Char, Dennis Gannon, Thomas T. Hewett, Chris Johnson, and John R. Rice. 2000. Workshop on scientific knowledge, information and computing (SIDEKI'98). *Enabling Technol. Comput. Sci.: Framew. Middlew. Environ.* 548 (2000), 19.

[20] Grant R. Brammer, Ralph W. Crosby, Suzanne J. Matthews, and Tiffani L. Williams. 2011. Paper Mâché: Creating dynamic reproducible science. *Proced. Comput. Sci.* 4 (2011), 658–667.
[21] Tim Bray, Jean Paoli, C. Michael Sperberg-McQueen, Eve Maler, and Franois Yergeau. 1997. Extensible Markup Language (XML). *World Wide Web Journal* 2, 4 (1997), 27–66.
[22] John Bresnahan, Tim Freeman, David LaBissoniere, and Kate Keahey. 2011. Managing appliance launches in infrastructure clouds. In *Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery*. ACM, 12.
[23] Eric A. Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of the 6th ACM Symposium on Cloud Computing*. ACM, 167–167.
[24] Tomasz Buchert, Cristian Ruiz, Lucas Nussbaum, and Olivier Richard. 2015. A survey of general-purpose experiment management tools for distributed systems. *Future Gener. Comput. Syst.* 45 (2015), 1–12.
[25] Jonathan B. Buckheit and David L. Donoho. 1995. *Wavelab and Reproducible Research*. Springer.
[26] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *Database Theory—ICDT 2001*. Springer, 316–330.
[27] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. 2006. VisTrails: Visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. ACM, 745–747.
[28] Franck Cappello, Eddy Caron, Michel Dayde, Frédéric Desprez, Yvon Jégou, Pascale Primet, Emmanuel Jeannot, Stéphane Lanteri, Julien Leduc, Nouredine Melab, et al. 2005. Grid'5000: A large scale and highly reconfigurable grid experimental testbed. In *Proceedings of the 6th IEEE/ACM International Workshop on Grid Computing*. IEEE Computer Society, 99–106.
[29] Dylan Chapp, Travis Johnston, and Michela Taufer. 2015. On the need for reproducible numerical accuracy through intelligent runtime selection of reduction algorithms at the extreme scale. In *Proceedings of the 2015 IEEE International Conference on Cluster Computing (CLUSTER'15)*, IEEE, 166–175.
[30] Fernando Chirigati, Dennis Shasha, and Juliana Freire. 2013. Reprozip: Using provenance to support computational reproducibility. In *Presented as Part of the 5th USENIX Workshop on the Theory and Practice of Provenance*.
[31] CircleCI. 2017. Continuous Integration and Delivery—CircleCI. Retrieved August 2, 2017, https://circleci.com/.
[32] Jon Claerbout. 2011. Making Scientific Contributions Reproducible. Retrieved July 11, 2006, http://sepwww.stanford.edu/oldsep/matt/join/redoc/web/iris.html.
[33] Jon Claerbout and Martin Karrenbach. 1992. Electronic documents give reproducible research a new meaning. In *Proceedings of the 62nd Annual International Meeting of the Society of Exploration Geophysics*. 601–604.
[34] J. F. Claerbout. 1991. *Electronic Document Preface*. Technical Report SEP-72. Stanford Exploration Project. 18 pages. http://sepwww.stanford.edu/public/docs/sep72/jon3/paper_html/node4.html.
[35] National Research Council et al. 2012. *Assessing the Reliability of Complex Models: Mathematical and Statistical Foundations of Verification, Validation, and Uncertainty Quantification*. National Academies Press.
[36] Ludovic Courtès and Ricardo Wurmus. 2015. Reproducible and user-controlled software environments in HPC with guix. In *European Conference on Parallel Processing*. Springer, 579–591.
[37] Jennifer Crocker and M. Lynne Cooper. 2011. Addressing scientific fraud. *Science* 334, 6060 (2011), 1182–1182.
[38] Donald Dabdub, K. Mani Chandy, and Thomas T. Hewett. 2000. Managing specificity and generality: Tailoring general archetypal PSEs to specific users. In *Enabling Technologies for Computational Science*. Springer, 65–77.
[39] Andrew Davison. 2012. Automated capture of experiment context for easier reproducibility in computational research. *Comput. Sci. Eng.* 14, 4 (2012), 48–56.
[40] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. 2004. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*. Springer, 11–20.
[41] Ewa Deelman, Dennis Gannon, Matthew Shields, and Ian Taylor. 2009. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Gener. Comput. Syst.* 25, 5 (2009), 528–540.
[42] David Devecsery, Michael Chow, Xianzheng Dou, Jason Flinn, and Peter M. Chen. 2014. Eidetic systems. In *Proceedings of the 11th USENIX Symposium on Oper. Systems Design and Implementation (OSDI'14)*, Vol. 14. 525–540.
[43] Paolo Di Tommaso, Evan Floden, Maria Chatzou, and Cedric Notredame. 2017. Using the NextFlow framework for reproducible in-silico omics analyses across clusters and clouds. *PeerJ Preprints* 5 (2017), e2796v1.
[44] Andrew Dienstfrey and Ronald Boisvert. 2012. *Uncertainty Quantification in Scientific Computing: 10th IFIP WG 2.5 Working Conference (WoCoUQ'11)*, Vol. 377. Springer.
[45] Christian Dietrich and Daniel Lohmann. 2015. The dataref versuchung: Saving time through better internal repeatability. *ACM SIGOPS Oper. Systems Rev.* 49, 1 (2015), 51–60.
[46] Eelco Dolstra and Andres Löh. 2008. NixOS: A purely functional Linux distribution. In *ACM Sigplan Not.*, Vol. 43. ACM, 367–378.
[47] Carsten Dominik. 2010. *The Org Mode 7 Reference Manual-Organize your life with GNU Emacs*. Network Theory Ltd.
[48] Chris Drummond. 2009. Replicability is not reproducibility: Nor is it good science. Cogprints Technical Report #7691. http://cogprints.org/7691/7/ICMLws09.pdf.

[49] Marlon Dumas and Arthur H. M. Ter Hofstede. 2001. UML activity diagrams as a workflow specification language. In *Proceedings of the International Conference on the Unified Modeling Language*. Springer, 76–90.
[50] Paul M. Duvall. 2007. *Continuous Integration*. Pearson Education India.
[51] Sarah Edwards, Xuan Liu, and Niky Riga. 2015. Creating repeatable computer science and networking experiments on shared, public testbeds. *ACM SIGOPS Oper. Systems Rev.* 49, 1 (2015), 90–99.
[52] Bo Einarsson. 2005. *Accuracy and Reliability in Scientific Computing*. SIAM.
[53] Joseph Emeras, Bruno Bzeznik, Olivier Richard, Yiannis Georgiou, and Cristian Ruiz. 2012. Reconstructing the software environment of an experiment with Kameleon. In *Proceedings of the 5th ACM COMPUTE Conference: Intelligent & Scalable System Technologies*. ACM, 16.
[54] Hakan Erdogmus, Maurizio Morisio, and Marco Torchiano. 2005. On the effectiveness of test-first approach to programming. In *IEEE Transactions on Software Engineering* 31, 3 (2005), 226–237.
[55] Dror G. Feitelson. 2015. From repeatability to reproducibility and corroboration. *ACM SIGOPS Oper. Syst. Rev.* 49, 1 (2015), 3–11.
[56] Karl Fogel. 2005. *Producing Open Source Software: How to Run a Successful Free Software Project*. O'Reilly Media, Inc.
[57] Sergey Fomel. 2015. Reproducible research as a community effort: Lessons from the madagascar project. *Comput. Sci. Eng.* 17, 1 (2015), 20–26.
[58] Sergey Fomel and Gilles Hennenfent. 2007. Reproducible computational experiments using scons. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'07)*, Vol. 4. IEEE, IV–1257.
[59] Sergey Fomel, Paul Sava, Ioan Vlad, Yang Liu, and Vladimir Bashkardin. 2013. Madagascar: Open-source software project for multidimensional data analysis and reproducible computational experiments. *J. Open Res. Softw.* 1, 1 (2013).
[60] Sergey Fomel, Matthias Schwab, and Joel Schroeder. 1997. Empowering SEP's documents. *SEP-94: Stanford Exploration Project* (1997), 339–361.
[61] Ian Foster, Jens Vockler, Michael Wilde, and Yong Zhao. 2002. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of the 14th International Conference on the Scientific and Statistical Database Management, 2002*. IEEE, 37–46.
[62] Martin Fowler and Matthew Foemmel. 2006. Continuous integration. *Thought-Works*, Retrieved from http://www.thoughtworks.com/ContinuousIntegration.pdf, 122.
[63] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. 2008. Provenance for computational tasks: A survey. *Comput. Sci. Eng.* 10, 3 (2008).
[64] James Frey. 2002. Condor DAGMan: Handling inter-job dependencies. Technical report, University of Wisconsin, Dept. of Computer Science).
[65] Hideaki Fuju, Shigeo Kawata, Hideaki Sugiura, Yuichi Saitoh, Yoshikazu Hayase, Hitohide Usami, Motohiro Yamada, Yutaka Miyahara, Hiroyuki Kanazawa, and Takashi Kikuchi. 2006. Scientific simulation execution support on a closed distributed computer environment. In *Proceedings of the 2nd IEEE International Conference on e-Science and Grid Computing (e-Science'06)*. IEEE, 109–109.
[66] Efstratios Gallopoulos, Elias Houstis, and John R. Rice. 1994. Computer as thinker/doer: Problem-solving environments for computational science. *IEEE Comput. Sci. Eng.* 1, 2 (1994), 11–23.
[67] John R. Rice. 1991. Future research directions in problem solving environments for computational science. In *Proceedings of the IFIP TC2/WG 2.5 Working Conference on Programming Environments for High-Level Scientific Problem Solving*. North-Holland Publishing Co., 363–369.
[68] Rogel Garcia and Marco Tulio Valente. NextFlow: Business process meets mapping frameworks. Retrieved March 9, 2017, http://www.nextflow.org/downloads/Nextflow_tech_report.pdf.
[69] Daniel Garijo, Oscar Corcho, and Yolanda Gil. 2013. Detecting common scientific workflow fragments using templates and execution provenance. In *Proceedings of the 7th International Conference on Knowledge Capture*. ACM, 33–40.
[70] Matan Gavish and David Donoho. 2011. A universal identifier for computational results. *Proced. Comput. Sc.* 4 (2011), 637–647.
[71] Belinda Giardine, Cathy Riemer, Ross C. Hardison, Richard Burhans, Laura Elnitski, Prachi Shah, Yi Zhang, Daniel Blankenberg, Istvan Albert, James Taylor, et al. 2005. Galaxy: A platform for interactive large-scale genome analysis. *Genome Res.* 15, 10 (2005), 1451–1455.
[72] Carole Goble. 2013. Results may vary. Reproducibility, open science, and all that jazz (July 2013). Keynote given by Carole Goble on July 23, 2013 at ISMB/ECCB 2013. Retrieved November 9, 2016, http://www.slideshare.net/carolegoble/ismb2013-keynotecleangoble/17.
[73] O. S. Gómez, N. Juristo, and S. Vegas. 2010. Replication, reproduction and re-analysis: Three ways for verifying experimental findings. In *Proceedings of the 1st International Workshop on Replication in Empirical Software Engineering Research (RESER'10), Cape Town, South Africa*.

[74] Alyssa Goodman, Alberto Pepe, Alexander W. Blocker, Christine L. Borgman, Kyle Cranmer, Merce Crosas, Rosanne Di Stefano, Yolanda Gil, Paul Groth, Margaret Hedstrom, et al. 2014. Ten simple rules for the care and feeding of scientific data. *PLoS Comput. Biol.* 10, 4 (2014), e1003542.
[75] Eelco Dolstra and Eelco Visser. 2007. Automated software testing and release with nix build farms. In *Proceedings of the 3rd European Symposium on Verification and Validation of Software Systems (VVSS'07)*. Eindhoven University of Technology, 65–77.
[76] Zhijie Guan, Francisco Hernandez, Purushotham Bangalore, Jeff Gray, Anthony Skjellum, Vijay Velusamy, and Yin Liu. 2006. Grid-flow: A grid-enabled scientific workflow system with a petri-net-based interface. *Concurr. Comput.: Pract. Exp.* 18, 10 (2006), 1115–1140.
[77] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, Vol. 10. 1–8.
[78] Mitchell Hashimoto. 2013. *Vagrant: Up and Running*. O'Reilly Media, Inc.
[79] Les Hatton and Gregory Warr. 2016. Full computational reproducibility in biological science: Methods, software and a case study in protein biology. *arXiv:1608.06897* (2016).
[80] Francisco Hernández, Purushotham Bangalore, Jeff Gray, and Kevin Reilly. 2005. A graphical modeling environment for the generation of workflows for the globus toolkit. In *Component Models and Systems for Grid Applications*. Springer, 79–96.
[81] Thomas T. Hewett and Jennifer L. DePaul. 2000. Toward a human centered scientific problem solving environment. In *Kluwer International Series in Engineering and Computer Science*. 79–90.
[82] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, Vol. 11, 22–22.
[83] Andreas Hoheisel. 2006. User tools and languages for graph-based Grid workflows. *Concurr. Comput.: Pract. Exp.* 18, 10 (2006), 1101–1113.
[84] David Hollingsworth and U. K. Hampshire. 1995. Workflow management coalition: The workflow reference model. *Document Number TC00-1003* 19 (1995). http://www.pa.icar.cnr.it/cossentino/ICT/doc/D12.1%20-%20Workflow%20Management%20Coalition%20-%20The%20Workflow%20Reference%20Model.pdf.
[85] Elias N. Houstis, John R. Rice, Efstratios Gallopoulos, and Randall Bramley. 2012. *Enabling Technologies for Computational Science: Frameworks, Middleware and Environments*, Vol. 548. Springer Science & Business Media.
[86] Bill Howe. 2012. CDE: A tool for creating portable experimental software packages. *Comput. Sci. Eng.* 14, 4 (2012), 32–35.
[87] Bill Howe. 2012. Virtual appliances, cloud computing, and reproducible research. *Comput. Sci. Eng.* 14, 4 (2012), 36–41.
[88] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R. Pocock, Peter Li, and Tom Oinn. 2006. Taverna: A tool for building and running workflows of services. *Nucleic Acids Res.* 34, Suppl. 2 (2006), W729–W732.
[89] John P. A. Ioannidis. 2005. Why most published research findings are false. *PLoS Med* 2, 8 (2005), e124.
[90] Peter Ivie and Douglas Thain. 2016. PRUNE: A preserving run environment for reproducible scientific computing. In *Proceedings of the IEEE Conference on e-Science*.
[91] P. Ivie, C. Zheng, and D. Thain. 2016. An analysis of reproducibility and non-determinism in HEP software and ROOT data. In *J. Phys.: Conf. Ser.* IOP Publishing.
[92] Barbara R. Jasny, Gilbert Chin, Lisa Chong, and Sacha Vignieri. 2011. Again, and again, and again? *Science* 334, 6060 (2011), 1225–1225.
[93] Emmanuel Jeanvoine, Luc Sarzyniec, and Lucas Nussbaum. 2013. Kadeploy3: Efficient and scalable operating system provisioning for clusters. *USENIX; Login:* 38, 1 (2013), 38–44.
[94] Jenkins. 2017. Jenkins. Retrieved August 2, 2017, https://jenkins.io/.
[95] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2017. The popper convention: Making reproducible systems evaluation practical. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'17)*. IEEE, 1561–1570.
[96] Chris Johnson. 2004. Top scientific visualization research problems. *IEEE Comput. Graph. Appl.* 24, 4 (2004), 13–17.
[97] Shigeo Kawata. 2015. Computer assisted problem solving environment (PSE). In *Encyclopedia of Information Science and Technology* (3rd ed.). IGI Global, 1251–1260.
[98] Jihie Kim, Ewa Deelman, Yolanda Gil, Gaurang Mehta, and Varun Ratnakar. 2008. Provenance trails in the wings/pegasus system. *Concurr. Computat.: Pract. Exp.* 20, 5 (2008), 587–597.
[99] Jonathan Klinginsmith, Malika Mahoui, and Yuqing Melanie Wu. 2011. Towards reproducible escience in the cloud. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom'11)*. IEEE, 582–586.
[100] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter notebooks? A publishing format for

reproducible computational workflows. *Positioning and Power in Academic Publishing: Players, Agents and Agendas* (2016), 87.
[101] Steven Knight. 2005. Building software with SCons. *Comput. Sci. Eng.* 7, 1 (2005), 79–88.
[102] Ivan Krsul, Arijit Ganguly, Jian Zhang, Jose A. B. Fortes, and Renato J. Figueiredo. 2004. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *Proceedings of the ACM/IEEE SC2004 Conference on Supercomputing, 2004.* IEEE, 7–7.
[103] Christine Laine, Steven N. Goodman, Michael E. Griswold, and Harold C. Sox. 2007. Reproducible research: Moving toward research the public can really trust. *Ann. Intern. Med.* 146, 6 (2007), 450–453.
[104] Dag Toppe Larsen, Jakob Blomer, Predrag Buncic, Ioannis Charalampidis, and Artem Haratyunyan. 2012. Long-term preservation of analysis software environment. In *J. Phys.: Conf. Ser.*, Vol. 396. IOP Publishing, 032064.
[105] Yung-Li Lee, Mark E. Barkey, and Hong-Tae Kang. 2011. *Metal Fatigue Analysis Handbook: Practical Problem-Solving Techniques for Computer-Aided Engineering.* Elsevier.
[106] Jeffrey T. Leek and Roger D. Peng. 2015. Opinion: Reproducible research can still be wrong: Adopting a prevention approach. *Proceedings of the National Academy of Sciences* 112, 6 (2015), 1645–1646.
[107] Randall J. LeVeque, Ian M. Mitchell, and Victoria Stodden. 2012. Reproducible research for scientific computing: Tools and strategies for changing the culture. *Comput. Sci. Eng.* 14, 4 (2012), 13.
[108] Frank Leymann et al. 2001. Web Services FlowLanguage (WSFL 1.0). (2001).
[109] Ji Liu, Esther Pacitti, Patrick Valduriez, and Marta Mattoso. 2015. A survey of data-intensive scientific workflow management. *J. Grid Comput.* 13, 4 (2015), 457–493.
[110] Jon Loeliger. 2006. Collaborating with GIT. *Linux Mag.* June (2006).
[111] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C. Webb, and Ken Yocum. 2010. Stateful bulk processing for incremental analytics. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 51–62.
[112] James Loope. 2011. *Managing Infrastructure with Puppet.* O'Reilly Media, Inc.
[113] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A. Lee, Jing Tao, and Yang Zhao. 2006. Scientific workflow management and the Kepler system. *Concurr. Comput.: Pract. Exp.* 18, 10 (2006), 1039–1065.
[114] Bertram Ludascher, Ilkay Altintas, and Amarnath Gupta. 2003. Compiling abstract scientific workflows into web service workflows. In *Proceedings of the 15th International Conference on Scientific and Statistical Database Management, 2003.* IEEE, 251–254.
[115] Cory Lueninghoener. 2011. Getting started with configuration management. (2011).
[116] Ben Marwick. 2016. Computational reproducibility in archaeological research: Basic principles and a case study of their implementation. *J. Archaeol. Meth. Theor.* (2016), 1–27.
[117] Anthony Mayer, Steve McGough, Nathalie Furmento, William Lee, Steven Newhouse, and John Darlington. 2003. ICENI dataflow and workflow: Composition and scheduling in space and time. In *UK e-Science All Hands Meeting*, Vol. 634. 627.
[118] Robert Mecklenburg. 2004. *Managing Projects with GNU Make.* O'Reilly Media, Inc.
[119] Haiyan Meng and Douglas Thain. 2015. 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, New York, NY.
[120] Dirk Merkel. 2014. Docker: Lightweight linux containers for consistent development and deployment. *Linux J.* 2014, 239 (2014), 2.
[121] Ralph C. Merkle. 1982. Method of providing digital signatures. (Jan. 5 1982). US Patent 4,309,569. File date: Sep. 5, 1979.
[122] Jill P. Mesirov. 2010. Accessible reproducible research. *Science* 327, 5964 (2010), 415–416.
[123] Steffen Meyer, Patrick Healy, Theo Lynn, and Jim Morrison. 2013. Quality assurance for open source software configuration management. In *Proceedings of the 2013 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC'13)*, IEEE, 454–461.
[124] Roger E. Millsap and Howard T. Everson. 1993. Methodology review: Statistical approaches for assessing measurement bias. *Appl. Psychol. Meas.* 17, 4 (1993), 297–334.
[125] Gyöngyvér Molnár and Benő Csapó. 2017. Exploration and learning strategies in an interactive problem-solving environment at the beginning of higher education studies. (2017).
[126] Kevin Murrell. 2013. The Harwell Dekatron computer. In *Making the History of Computing Relevant.* Springer, 309–313.
[127] James Myers, Margaret Hedstrom, Dharma Akmon, Sandy Payette, Beth A. Plale, Inna Kouper, Scott McCaulay, Robert McDonald, Isuru Suriarachchi, Aravindh Varadharaju, et al. 2015. Towards sustainable curation and preservation: The SEAD project's data services approach. In *Proceedings of the 2015 IEEE 11th International Conference on e-Science (e-Science'15)*, IEEE, 485–494.

[128] Chris J. Oates, Jim Q. Smith, and Sach Mukherjee. 2016. Estimating causal structure using conditional DAG models. *J. Mach. Learn. Res.* 17, 54 (2016), 1–23.
[129] William L. Oberkampf and Christopher J. Roy. 2010. *Verification and Validation in Scientific Computing*. Cambridge University Press.
[130] Tom Oinn, Matthew Addis, Justin Ferris, Darren Marvin, Martin Senger, Mark Greenwood, Tim Carver, Kevin Glover, Matthew R. Pocock, Anil Wipat, et al. 2004. Taverna: A tool for the composition and enactment of bioinformatics workflows. *Bioinformatics* 20, 17 (2004), 3045–3054.
[131] Tom Oinn, Mark Greenwood, Matthew Addis, M. Nedim Alpdemir, Justin Ferris, Kevin Glover, Carole Goble, Antoon Goderis, Duncan Hull, Darren Marvin, et al. 2006. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurr. Comput.: Pract. Exp.* 18, 10 (2006), 1067–1100.
[132] Sudhir Pandey. 2012. Investigating community, reliability and usability of CFEngine, Chef and Puppet. Master thesis. University of Oslo's Department of Informatics.
[133] Roger Peng. 2015. The reproducibility crisis in science: A statistical counterattack. *Significance* 12, 3 (2015), 30–32.
[134] Roger D. Peng. 2011. Reproducible research in computational science. *Science* 334, 6060 (2011), 1226–1227.
[135] Quan Pham, Tanu Malik, and Ian Foster. 2013. Using provenance for repeatability. In *Presented as Part of the 5th USENIX Workshop on the Theory and Practice of Provenance*. 5–8.
[136] Karl Popper. 2005. *The Logic of Scientific Discovery*. Routledge.
[137] Florian Prinz, Thomas Schlange, and Khusru Asadullah. 2011. Believe it or not: How much can we rely on published data on potential drug targets? *Nat. Rev. Drug Discov.* 10, 9 (2011), 712–712.
[138] Todd Proebsting, Alex M. Warren, and Christian Collberg. 2015. Repeatability and benefaction in computer systems research. *University of Arizona TR 14*. Vol. 4. 1–68.
[139] Min Ragan-Kelley, F. Perez, B. Granger, T. Kluyver, P. Ivanov, J. Frederic, and M. Bussonier. 2014. The jupyter/IPython architecture: A unified view of computational research, from interactive exploration to communication and publication. In *AGU Fall Meeting Abstracts*, Vol. 1, 07.
[140] Arcot Rajasekar, Reagan Moore, Chien-Yi Hou, Christopher A. Lee, Richard Marciano, Antoine de Torcy, Michael Wan, Wayne Schroeder, Sheau-Yen Chen, Lucas Gilbert, et al. 2010. iRODS primer: Integrated rule-oriented data system. *Synth. Lect. Inform. Concepts, Retr. Serv.* 2, 1 (2010), 1–143.
[141] Joyce M. Ray. 2014. *Research Data Management: Practical Strategies for Information Professionals*. Purdue University Press.
[142] John R. Rice. 2000. Future challenges for scientific simulation. In *Enabling Technologies for Computational Science*. Springer, 7–17.
[143] Cristian Ruiz, Olivier Richard, and Joseph Emeras. 2014. Reproducible software appliances for experimentation. In *Testbeds and Research Infrastructure: Development of Networks and Communities*. Springer, 33–42.
[144] James Rumbaugh, Ivar Jacobson, and Grady Booch. 2004. *The Unified Modeling Language Reference Manual*. Pearson Higher Education.
[145] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. 1999. Deciding when to forget in the elephant file system. In *ACM SIGOPS Oper. Syst. Rev.* 33. ACM, 110–123.
[146] Matthias Schwab, Martin Karrenbach, and Jon Claerbout. 2000. Making scientific computations reproducible. *Comput. Sci. Eng.* 2, 6 (2000), 61–67.
[147] Barbara Sierman. 2014. The SCAPE policy framework, maturity levels and the need for realistic preservation policies. *IPRES 2014 Proceedings* 259.
[148] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. 2005. A survey of data provenance in e-science. *ACM Sigmod Rec.* 34, 3 (2005), 31–36.
[149] Munindar P. Singh and Mladen A. Vouk. 1996. Scientific workflows: Scientific computing meets transactional workflows. In *Proceedings of the NSF Workshop on Workflow and Process Automation in Information Systems: State-of-the-Art and Future Directions*. 28–34.
[150] Luka Stanisic, Arnaud Legrand, and Vincent Danjean. 2015. An effective git and org-mode based workflow for reproducible research. *ACM SIGOPS Oper. Syst. Rev.* 49, 1 (2015), 61–70.
[151] Victoria Stodden. 2011. Trust your science? Open your data and code. *Amstat News* (2011), 21–22.
[152] Victoria Stodden, Jonathan Borwein, and David H. Bailey. 2013. Setting the default to reproducible. *Comput. Sci. Res. SIAM News* 46 (2013), 4–6.
[153] Victoria Stodden, Friedrich Leisch, and Roger D. Peng. 2014. *Implementing Reproducible Research*. CRC Press.
[154] Victoria Stodden and Sheila Miguez. 2013. Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. *Available at SSRN 2322276* (2013).
[155] Sam Sun, Larry Lannom, and Brian Boesch. 2003. *Handle System Overview*. Technical Report. The Internet Society.
[156] Martin Szomszor and Luc Moreau. 2003. Recording and reasoning over data provenance in web and grid services. In *On the Move to Meaningful Internet Syst. 2003: CoopIS, DOA, and ODBASE*. Springer, 603–620.

[157] Ian Taylor, Matthew Shields, Ian Wang, and Andrew Harrison. 2007. The triana workflow environment: Architecture and applications. In *Workflows for e-Science*. Springer, 320–339.
[158] Ian J. Taylor, Ewa Deelman, Dennis B. Gannon, and Matthew Shields. 2014. *Workflows for e-Science: Scientific Workflows for Grids*. Springer.
[159] Mischa Taylor and Seth Vargo. 2014. *Learning Chef: A Guide to Configuration Management and Automation*. O'Reilly Media, Inc.
[160] Takayuki Teramoto, Tadashi Okada, and Shigeo Kawata. 2007. A distributed education-support PSE system. In *IEEE International Conference on e-Science and Grid Computing*. IEEE, 516–520.
[161] Travis CI. 2017. Travis CI—Test and Deploy Your Code with Confidence. Retrieved August 2, 2017, https://travis-ci.org/.
[162] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. 2007. Opium: Optimal package install/uninstall manager. In *Proceedings of the 29th International Conference on Software Engineering*. IEEE Computer Society, 178–188.
[163] Matthew J. Turk. 2013. Scaling a code in the human dimension. In *Proceedings of the Conference on Extreme Science and Engineering Discovery Environment: Gateway to Discovery*. ACM, 69.
[164] Amin Vahdat and Thomas E. Anderson. 1998. Transparent result caching. In *USENIX Annual Technical Conference*.
[165] Wil Van Der Aalst and Kees Max Van Hee. 2004. *Workflow Management: Models, Methods, and Systems*. MIT Press.
[166] Wil M. P. Van der Aalst. 1998. The application of petri nets to workflow management. *J. Circuits, Syst. Comput.* 8, 01 (1998), 21–66.
[167] Wil M. P. Van Der Aalst and Arthur H. M. Ter Hofstede. 2005. YAWL: Yet another workflow language. *Inform. Syst.* 30, 4 (2005), 245–275.
[168] Sander Van Der Burg, Merijn de Jonge, Eelco Dolstra, and Eelco Visser. 2009. Software deployment in a dynamic cloud: From device to service orientation in a hospital environment. In *Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing*. IEEE Computer Society, 61–66.
[169] Mayank Varia, Benjamin Price, Nicholas Hwang, Ariel Hamlin, Jonathan Herzog, Jill Poland, Michael Reschly, Sophia Yakoubov, and Robert K. Cunningham. 2015. Automated assessment of secure search systems. *ACM SIGOPS Oper. Syst. Rev.* 49, 1 (2015), 22–30.
[170] H. M. W. Verbeek, Alexander Hirnschall, and Wil M. P. van der Aalst. 2002. XRL/flower: Supporting inter-organizational workflows using XML/Petri-net technology. In *International Workshop on Web Services, E-Business, and the Semantic Web*. Springer, 93–108.
[171] Gregor Von Laszewski, Mihael Hategan, and Deepti Kodeboyina. 2007. Java CoG kit workflow. In *Workflows for e-Science*. Springer, 340–356.
[172] Greg Wilson, Dhavide A. Aruliah, C. Titus Brown, Neil P. Chue Hong, Matt Davis, Richard T. Guy, Steven HD Haddock, Kathryn D. Huff, Ian M. Mitchell, Mark D. Plumbley, et al. 2014. Best practices for scientific computing. *PLoS Biol.* 12, 1 (2014), e1001745.
[173] Roundtable Participants Yale. 2010. Reproducible research. *Compu. Sci. Eng.* 12, 5 (2010), 8–13.
[174] Jia Yu and Rajkumar Buyya. 2004. A novel architecture for realizing grid workflow using tuple spaces. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*. IEEE, 119–128.
[175] Jia Yu and Rajkumar Buyya. 2005. A taxonomy of workflow management systems for grid computing. *J. Grid Comput.* 3, 3–4 (2005), 171–200.
[176] Xiang Zhao, Emery R. Boose, Yuriy Brun, Barbara Staudt Lerner, and Leon J. Osterweil. 2013. Supporting undo and redo in scientific data analysis. In *TaPP*.

Received March 2017; revised January 2018; accepted February 2018