Facilitating the Reproducibility of Scientific Workflows with Execution Environment Specifications

Haiyan Meng and Douglas Thain

Department of Computer Science and Engineering, University of Notre Dame
Notre Dame, Indiana, USA
hmeng@nd.edu and dthain@nd.edu

Abstract

Scientific workflows are designed to solve complex scientific problems and accelerate scientific progress. Ideally, scientific workflows should improve the reproducibility of scientific applications by making it easier to share and reuse workflows between scientists. However, scientists often find it difficult to reuse others' workflows, which is known as *workflow decay*. In this paper, we explore the challenges in reproducing scientific workflows, and propose a framework for facilitating the reproducibility of scientific workflows at the task level by giving scientists complete control over the execution environments of the tasks in their workflows and integrating execution environment specifications into scientific workflow systems. Our framework allows dependencies to be archived in basic units of OS image, software and data instead of gigantic all-in-one images. We implement a prototype of our framework by integrating *Umbrella*, an execution environment creator, into *Makeflow*, a scientific workflow system.

To evaluate our framework, we use it to run two bioinformatics scientific workflows, *BLAST* and *BWA*. The execution environment of the tasks in each workflow is specified as an Umbrella specification file, and sent to execution nodes where *Umbrella* is used to create the specified environment for running the tasks. For each workflow we evaluate the size of the Umbrella specification file, the time and space overheads of creating execution environments using *Umbrella*, and the heterogeneity of execution nodes contributing to each workflow. The evaluation results show that our framework improves the utilization of heterogeneous computing resources, and improves the portability and reproducibility of scientific workflows.

Keywords: reproducible research, scientific workflows, execution environment specifications

1 Introduction

The reproducibility of scientific applications has become increasingly important for the progress of computational science because it allows the original author and others to reproduce, verify, and further extend the original applications [10]. Different solutions have been proposed to

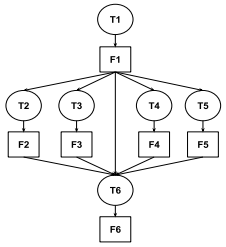

Figure 1: An Example Makeflow in DAG

```
F6 capitol.montage.gif: capitol.jpg capitol.90.jpg capitol.180.jpg
   capitol.270.jpg capitol.360.jpg /usr/bin/convert                    T6
           convert -delay 10 -loop 0 capitol.jpg capitol.90.jpg
   capitol.180.jpg capitol.270.jpg capitol.360.jpg capitol.270.jpg
   capitol.180.jpg capitol.90.jpg capitol.montage.gif
F5 capitol.90.jpg: capitol.jpg                                         T5
           convert -swirl 90 capitol.jpg capitol.90.jpg
F4 capitol.180.jpg: capitol.jpg                                        T4
           convert -swirl 180 capitol.jpg capitol.180.jpg
F3 capitol.270.jpg: capitol.jpg                                        T3
           convert -swirl 270 capitol.jpg capitol.270.jpg
F2 capitol.360.jpg: capitol.jpg                                        T2
           convert -swirl 360 capitol.jpg capitol.360.jpg
F1 capitol.jpg: /usr/bin/curl                                          T1
           curl -o capitol.jpg http://ccl.cse.nd.edu/images/capitol.jpg
```

Figure 2: An Example Makefile: Image Rotation

reproduce single-machine scientific applications. Some popular solutions include virtual machines [9], Linux Containers (e.g., Docker [14]), and user-space ptrace-based tools (e.g., CDE [7] and Parrot-packaging tool [13]). In spite of their differences, these solutions all emphasize the importance of preserving the complete software stack (i.e., execution environment) of scientific applications for conducting reproducible research [12].

However, many scientific applications are too big to be solved on a single machine, due to their huge computing and storage requirements. To solve this, *scientific workflows* [16] were designed to disseminate complex data transformations and analysis procedures into a set of smaller and possibly independent tasks, which allows computing resources from clusters, grids and clouds to be utilized. The tasks involved in a scientific workflow are often organized into a directed acyclic graph (DAG), where nodes represents tasks and files, and edges represent data flow and dependency relationship. Figure 1 shows a DAG including six tasks, which represents the simple workflow example in Figure 2, written in the *Makefile* language [1]. A real scientific workflow is usually more complex in both task number and task dependencies.

To make it easy for scientists to compose and execute scientific workflows, a variety of *scientific workflow systems* have been developed [18], such as Taverna [15], Pegasus [4] and Makeflow [1]. The end-users of these workflow systems only need to specify a DAG of tasks. The workflow systems respond to communicate with execution engines, schedule tasks to the underlying computing resources, manage data sets and deliver fault tolerance.

Ideally, scientific workflows should improve the reproducibility of scientific applications by making it easier to share and reuse workflows between scientists. However, scientists often find it difficult to reuse others' workflows, which is known as *workflow decay* [8]. For example, a study in 2012 of Taverna workflows on myExperiment [6], a social website allowing scientists to share their workflows, shows that 80% of the workflows on the site cannot be reproduced [19].

Among the causes of workflow decay, the incompatible execution environments on execution nodes is a recurring significant problem [3, 8, 5, 2]. This work aims to improve the reproducibility of scientific workflows by bringing the incompatible execution environments to a minimum.

Depending on the scientific workflow system used, scientists have different levels of control over the underlying execution environments on execution nodes. Pegasus [4] allows scientists to compose *abstract workflows* without worrying about the details of the underlying execution environments, which means sysadmins must respond to the cumbersome job of configuring computing resources to meet all the requirements of different workflows. Makeflow [1] allows executables to be specified in workflow specifications and delivered to execution nodes, such as `/usr/bin/convert` in Figure 2. This is simple but not always correct, because executables may be sent to execution nodes with incompatible execution environments. To fix this, Makeflow allows scientists to specify a Docker image [14] containing the required execution environment, and delivers the image to execution nodes [20]. This gives scientists more control over the execution environments, but ends up with gigantic images which are expensive to store.

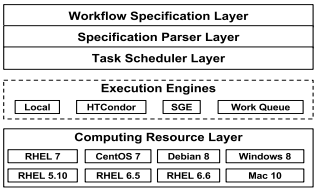

Figure 3: Layers of Scientific Workflow Systems

Attribute	Description
machine number	1886
hardware arch	x86_64, i686
kernel version	Linux, Darwin, Windows NT
OS	RHEL5, RHEL6, RHEL7, Mac9, Windows7
RHEL versions	5.11, 6.5, 6.6, 6.7, 6.8, 7.0, 7.2
CPU number	1, 2, 4, 8, 12, 16, 24, 32, 64
memory size	Min: 1002MB, Max: 251GB
disk size	Min: 9GB, Max: 4.7TB

Table 1: Heterogeneity of the ND HTCondor Pool (Nov. 2016)

In this paper, we explore the challenges in reproducing scientific workflows, and propose a framework for facilitating the reproducibility of scientific workflows at the task level by giving scientists complete control over the execution environments of the tasks in their workflows and integrating execution environment specifications into scientific workflow systems. Our framework allows dependencies to be archived in basic units of OS image, software and data instead of gigantic all-in-one images. We implement a prototype of our framework by integrating *Umbrella* [11], an execution environment creator, into *Makeflow* [1], a scientific workflow system which can utilize computing resources from a HTCondor pool [17].

To evaluate our framework, we use it to run two bioinformatics scientific workflows, *BLAST* and *BWA*. The execution environment of the tasks in each workflow is specified as an *Umbrella* specification file, and sent to the execution nodes in the Notre Dame HTCondor pool, where *Umbrella* is used to create the environment to run the tasks. For each workflow we evaluate the *Umbrella* specification file size, the time and space overheads of creating execution environments using *Umbrella*, and the heterogeneity of execution nodes contributing to each workflow.

2 Challenges in Reproducing Scientific Workflows

The reproducibility of scientific workflows depends on how scientific workflow systems are designed and implemented. In this section, we explore the characteristics of scientific workflow systems, which make it challenging to reproduce scientific workflows.

- **Complexity.** Scientific workflow systems usually include multiple layers, as shown in Figure 3. The complexity of different layers vary greatly. The workflow specification layer simply includes workflow languages and workflow specifications, which can be easily preserved. However, the task scheduler layer often communicates with multiple execution engines to achieve maximal speedup. The computing resource layer includes multiple software stacks (possibly thousands or even more), together with the networking connecting them together, and requires much more efforts to be preserved and reproduced.

- **Dynamics.** The stability of different layers of scientific workflow systems vary a lot. The workflow specification layer and the specification parser layer are usually very stable and tightly coupled. Adding new syntax into a workflow language would require the specification parser change accordingly. The task scheduler layer may add support for new execution engines as new computing frameworks become popular. The computing resource layer usually experiences more frequent changes, which is especially true for opportunistic computing framework like HTCondor [17].

- **Heterogeneity.** The hardware and software configurations of machines in the computing resource layer are often heterogeneous, both across different execution engines and within a single execution engine. For example, Table 1 shows the statistics of machine configurations of the Notre Dame HTCondor pool at November 2016, which is very different from

Executable (Shared Object Deps Num)	RHEL5.11	RHEL6.8	RHEL7.0
convert from RHEL5.11 (23)	✓	libMagick.so.10	libMagick.so.10
convert from RHEL6.8 (28)	libMagickCore.so.5	✓	liblcms.so.1
convert from RHEL7.0 (28)	libMagickCore.so.5	libtiff.so.5	✓

Table 2: Software Incompatibility across Different RHEL Distributions

the statistics captured at Spring 2015 [11]. It is worth noting that currently the Notre Dame HTCondor pool has 7 different versions of Red Hat Enterprise Linux (RHEL). Machines in cluster computing may start with homogeneous configurations, but would end up with heterogeneous configurations due to hardware replacement and upgrade.

- **Incompatible Execution Environments on Execution Nodes.** Execution nodes in the computing resource layer often do not have the correct execution environments required by tasks. One possible solution is to ask sysadmins to install all the missing dependencies on each execution node. However, this is not scalable due to the dependency complexity and diversity of different scientific workflows. Another solution is to allow scientists to specify executables and deliver them to execution nodes together with tasks, as done in Makeflow [1]. However, this is not always correct, even if we consider a very simple and common executable, `/usr/bin/convert`. Dynamically linked executables, such as `convert`, often have dozens of shared object dependencies, each of which has its own version and further dependencies. The numbers and versions of shared object dependencies of an executable on different OSes may vary a lot, as shown in the first column of Table 2. Therefore, sending an executable itself to execution nodes with different OSes often does not work due to the missing of dependencies. We ran `convert` from RHEL5-7 on RHEL5-7, and collected the results in Table 2. All the attempts of running an executable on a different OS version failed. We showed the first incompatible dependencies causing the failures in Table 2. The incompatible execution environment problem gets worse for real scientific workflows.

- **Hidden Network Dependencies.** Scientific workflows often have some data dependencies from third-party websites, which are obtained by `curl`-based or `wget`-based tasks, such as `T1` shown in Figure 2. When the workflow is tiny, these dependencies can be easily tracked. However, when the task number of a scientific workflow reaches 100s or 1000s, these network dependencies will be buried in the middle of the huge workflow specification and difficult to track. This may cause workflow decay if some fragile network dependencies are lost before being preserved properly.

3 A Framework Facilitating the Reproducibility of Scientific Workflows

Given the challenges of reproducing scientific workflows and the characteristics of scientific workflow systems, we propose a framework to facilitate the reproducibility of scientific workflows, which includes three parts:

- **Tracking Network Dependencies.** Instead of being hidden in the middle of scientific workflow specifications, network dependencies should be collected and tracked in a separate mechanism. For example, a file which maps data dependencies to their retrieval locations can be used to track the network dependencies as follows. This also makes it easy to redirect these dependencies if needed.

```
input       http://cse.research.org/blast/input
job.sched   http://cse.research.org/blast/job.sched
```

- **Specifying Execution Environments for Tasks.** A task in a workflow does not simply equal to the command line from the workflow specification, it has its own requirements of the underlying execution environment, which should include at least the information about the hardware configuration (hardware architecture, CPU, memory and disk), kernel type and version, OS name and version, the root filesystem image, software dependencies, data dependencies, command line and environment variables. The hardware and kernel information can be used to select the correct execution nodes from the underlying computing resources. The root filesystem image, software, data and environment variables information can be used to create the execution environment and execute the task in it.

  If all the tasks in a workflow share the same OS and software dependencies, a single execution environment specification can be composed and used to run all the tasks. When the dependencies of different tasks vary a lot in number or size, different execution environment specifications for different tasks can be used to avoid the space and time overheads of creating a huge execution environment on every execution node. To avoid composing a separate execution environment specification for every task, it is best to leave the intermediate data involved in a workflow out of execution environment specifications, and allow them to be specified by the specification parser layer automatically at runtime.

- **Sending Execution Environment Specifications to Execution Nodes.** To bring the incompatible execution environments on execution nodes to a minimum and improve the reproducibility of scientific workflows, instead of only sending the command line of a task to execution nodes, the execution environment specification of the task, together with the execution environment creator which can parse the specification and create the correct execution environment from it, should also be sent to execution nodes.

  To prepare executing a task, the execution environment creator obtains the root filesystem image, the software and data dependencies specified in an execution environment specification, creates a sandbox with sandbox techniques like virtual machines, Linux Containers and user-space tracers, and attaches the intermediate data into it. By doing this, instead of meet the execution environment requirements of different scientific workflows on all the execution nodes, scientific workflow systems only need to guarantee the execution environment creator itself can work on every execution node. This facilitates both the portability and reproducibility of scientific workflows.

4 An Implementation Prototype of The Framework

We implement a prototype of our framework by integrating *Umbrella*, an execution environment creator, into *Makeflow*, a scientific workflow system which can utilize computing resources from the local machine, SGE, Work Queue and HTCondor. To utilize Makeflow, scientists can compose their workflows in a workflow specification language called *Makefile* which is similar to *make*, as shown in Figure 2. In addition, a mount file which maps each network dependency to its retrieval location is introduced to track network dependencies.

Umbrella accepts a user-specified execution environment specification in JSON (such as *blast.umbrella* in Figure 4) and creates the specified environment using sandbox techniques like the Amazon EC2, Docker [14] and Parrot [13]. Instead of storing all the dependencies of a

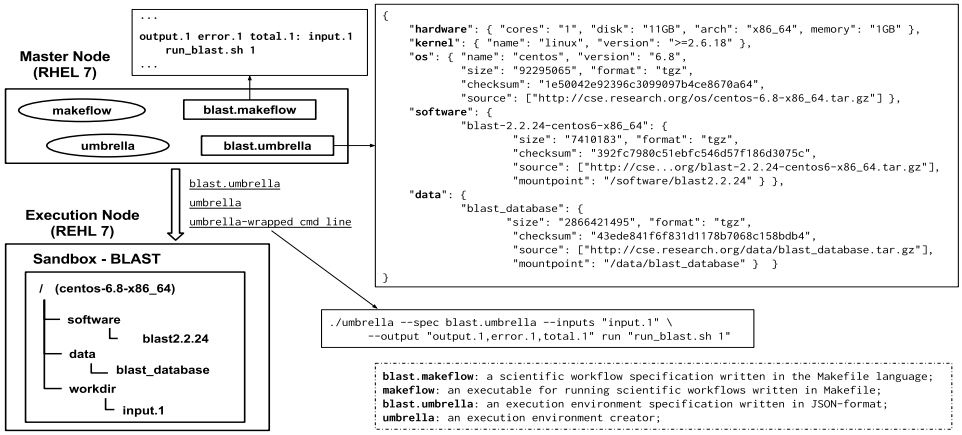

Figure 4: Running BLAST with Makeflow and Umbrella

task into a gigantic all-in-one package, *Umbrella* suggests the basic units of OS image, software and data. Relocatable software dependencies can be preserved and delivered as standalone packages, and non-relocatable software dependencies can be installed into an OS image and delivered with the OS image. This saves the storage space of archives and execution nodes by allowing common dependencies to be shared by different tasks. This also allows an execution environment to be specified in multiple sections like hardware, kernel and OS, which makes it easy to understand and reuse the environment. On each execution node, *Umbrella* creates a cache to store each unique dependency, allowing different tasks to share the same dependency from the cache concurrently.

Figure 4 shows how our framework can facilitate the reproducibility of a bioinformatics scientific workflow called *BLAST*, whose workflow specification is specified in a Makefile, *blast.makeflow*. Each task is expressed in a Makefile rule occupying two lines: the first line is in the format of `<outputs>:<inputs>`, and the second line `<cmd>`. Without our framework, scientists need to specify the executables needed for a task (such as `run_blast.sh`) in the `inputs` part of a rule, which is cumbersome and incomplete, because these executables may depend on other libraries and executables. Using our framework, an Umbrella specification, *blast.umbrella*, is composed to specify the execution environment information including hardware, kernel, OS, software and data dependencies needed by the tasks. The workflow specification, *blast.makeflow*, can focus on the inputs, outputs, and command line of each task.

Without our framework, the master node sends the command line of a task directly from workflow specifications to execution nodes, such as `run_blast.sh 1`, without any information about the expected execution environment. The tasks often fail on execution nodes due to incompatible execution environments. With our framework, the command line is wrapped into an Umbrella task, which brings together the execution environment specification, and the inputs, outputs and command line information from the workflow specification. The umbrella-wrapped command line, the execution environment specification, together with the execution environment creator, `umbrella`, are then sent to each execution node.

Umbrella is written in Python2.6. To run umbrella-wrapped tasks on execution nodes, the selection standards for the underlying computing resources should include the criteria for Python2.6. On execution nodes, an umbrella-wrapped task brings all the dependencies together, creates a sandbox, and runs the task inside it.

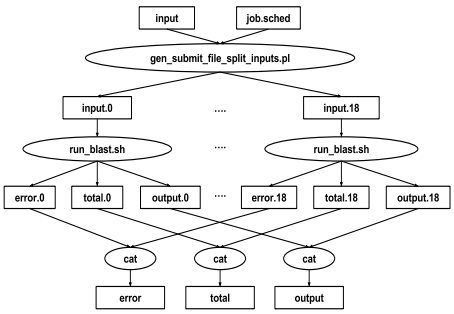

Figure 5: Workflow - BLAST

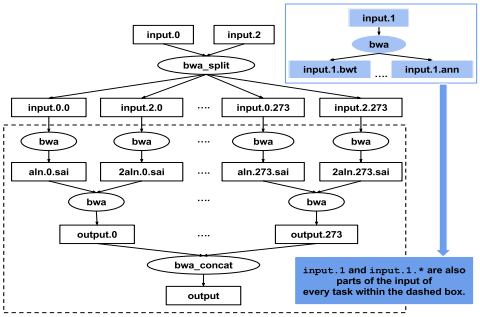

Figure 6: Workflow - BWA

4.1 Why not Use Docker as the Execution Environment Creator?

We tried to use Docker as the execution environment creator and run each task as a Docker container on execution nodes originally [20], and noticed the following limitations. First, running each task as a Docker container requires Docker to be installed on execution nodes and workers on execution nodes have permission to use Docker, both of which need root permission. Second, the size limit of a Docker container (by default is 10GB) exposes a limit on the storage space used by a task. Third, the storage drivers used by Docker usually have size limitations (e.g., 100GB used by *devicemapper*), which limits the number of Docker containers which can run concurrently on an execution node. Fourth, in the case where two different tasks pull the same Docker image or load Docker images from the same tarball, special concerns are required to solve the potential race condition. Fifth, it is not easy to track the metadata information of software and data dependencies, such as size, format and checksum, in *Dockerfile*. Sixth, a *Dockerfile* does not provide clear information to a new user about the location of software and data, who may want to run the Docker container with new software versions or date sets.

With Umbrella, each task can run as a Parrot job, which is a user-space ptrace-based tool and requires no root permission [13]. Umbrella does not have size limits on each task, and the total storage space Umbrella can utilize is only limited by the storage limit set by a worker. Unlike Docker trying to collect all the Docker images into a centralized storage space, such as `/var/lib/docker/devicemapper`, running each task as an Umbrella job only modifies the working directory of each worker. Umbrella organizes an execution environment into different sections (such as the `software` and `data` sections in Figure 4), each of which has its specific purpose. This allows these dependencies to be archived separately, and makes it very easy for a new user to test new software versions or new data sets.

5 Evaluation

To evaluate our framework, we specify the execution environments for two bioinformatics scientific workflows, *BLAST* (Figure 5) and *BWA* (Figure 6), as Umbrella specification files, and allow Makeflow to send the Umbrella specification files to execution nodes in the Notre Dame HTCondor pool, where the execution environment creator, `umbrella`, can be used to create the execution environment and run the task.

For each workflow we evaluate the size of the Umbrella execution environment specification, the space and time overheads of creating execution environments using Umbrella, and the heterogeneity of execution nodes contributing to each workflow.

Application	OS Dependency	Software Dependencies
BLAST	CentOS 6.8 (66MB/203MB)	perl (23MB/83MB), blast (7MB/22MB)
BWA	CentOS 6.8 (66MB/203MB)	perl (23MB/83MB), bwa (216KB/604KB)

Table 3: OS and Software Dependencies of Evaluated Workflows

Application	Umbrella Spec Size	Space Overhead	Time Overhead
BLAST	2.2KB	404MB	<1min
BWA	1.3KB	376MB	<1min

Table 4: Space and Time Overheads Introduced by Umbrella - OS and Software Deps

Except for the input and output dependencies embedded in workflow specifications, scientists can specify the underlying execution environments - including at least OS and software dependencies - for their workflows. Table 3 shows the OS and software dependencies of each evaluated workflow. The size information of each dependency in Table 3 is in the format of `tgz_format_compressed_size/uncompressed_size`. The relocatable software dependencies can be preserved and delivered as standalone packages. The non-relocatable software dependencies can be installed into the OS dependency via package managers and delivered as a fat OS dependency.

Our framework allows execution environment dependencies to be archived in fine granularity such as each basic root filesystem image, each software and each dataset, which saves the space overhead of archiving dependencies. For example, both *BLAST* and *BWA* depend on the same OS image and the same `perl` binary, as shown in Table 3. With our framework, the OS image and the `perl` binary only need to be archived once. As the number of scientific workflows depending on the same OS image or software binary increases, archiving dependencies in fine granularity becomes more advantageous and even necessary.

The *Umbrella Spec Size* column in Table 4 shows the size of the Umbrella execution environment specification file of each evaluated workflow, which is very tiny and should not cause any communication problem between the master node and the execution node. The *Space Overhead* column in Table 4 shows the space overhead of the OS and software dependencies introduced by running each task as an Umbrella job on an execution node, which includes both the compressed and uncompressed versions of each dependency. The *Time Overhead* column shows the time overhead introduced by obtaining these dependencies, uncompressing them if necessary, and integrating them into a unified sandbox. It is worth noting that the space and time overheads shown in Table 4 does not include the overhead of delivering the input files and the intermediate data of a scientific workflow to execution nodes, which are necessary even without our framework.

We ran each evaluated workflow with the features supported by our framework on the computing resources from the Notre Dame HTCondor pool. With our framework, the execution nodes with incompatible execution environments can be utilized by creating the required execution environments based on the Umbrella execution environment specifications sent by the master node. Table 5 shows the task number of each workflow, the number of tasks running on RHEL6, and the number of tasks running on RHEL7. Due to the preference settings on the computing resources in the HTCondor pool, all the tasks were scheduled onto the execution nodes with either RHEL6 or RHEL7. However, this does not mean these tasks can only run on RHEL6 or RHEL7. With our framework, these tasks can run successfully on any execution node satisfying the hardware and kernel requirements.

Application	Total Task Number	Tasks run on RHEL6	Tasks run on RHEL7
BLAST	23	15	8
BWA	825	366	459

Table 5: Heterogeneity of Execution Nodes Contributing to Each Workflow

6 Related Work

Different approaches have been utilized to facilitate the reproducibility of single-machine scientific applications. *Virtual machines* [9] wrap the whole execution environment of an application into a single file, which is easy to distribute but less feasible for complex applications with large software stacks. To decrease the space overhead of preserved artifacts, user-space tracers, such as CDE [7] and Parrot-packaging tool [13], trap the system calls of an application and only preserve the really used files. However, the context and structure of the preserved artifacts are not clear, which makes it difficult to reuse. To make it easy to reuse and extend scientific applications, Umbrella [12] takes the method of archiving the dependencies in finer granularity, and specifying the execution environments in an organized way. The specified execution environments can be (re)created with different sandbox techniques like virtual machines, Linux Containers (e.g., Docker [14]), and user-space tracers.

Due to the complexity of scientific workflows [5] and scientific workflow systems [16], the challenges in reproducing scientific workflows include but are not limited to the ones in reproducing single-machine scientific applications. Guidelines on how to design reproducible scientific workflows were proposed in [8, 5] to combat the *workflow decay* [8] problem, and *Research Objects* [2] were used to provide detailed metadata information about workflows [8]. However, these guidelines focus on the specification layer of scientific workflow systems and ignore the other components of scientific workflow systems [18]. The decay-parameters of the jobs in a scientific workflow are used to measure the possibility to reproduce the workflow and the cost to make it reproducible [3]. In this paper, we focus on how to improve the reproducibility of scientific workflows at the task level by giving scientists complete control over the execution environments of the tasks in their workflows and integrating execution environment specifications into scientific workflow systems.

7 Conclusion and Future Work

In this paper, we explore the challenges in reproducing scientific workflows, and propose a framework for facilitating the reproducibility of scientific workflows at the task level by integrating execution environment specifications into scientific workflow systems. Our framework improves the utilization of heterogeneous computing resources, and improves the portability and reproducibility of scientific workflows.

In the future work, we plan to explore how to reproduce distributed computing frameworks like HTCondor and how to reproduce scientific workflow systems, which include the master node scheduling the tasks and the underlying computing frameworks managing the worker nodes.

Acknowledgments

This work was supported in part by National Science Foundation grants PHY-1247316 (DASPOS), OCI-1148330 (SI2) and PHY-1312842. The Center for Research Computing at University of Notre Dame provided critical technical assistance throughout this research effort.

References

[1] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, page 1. ACM, 2012.

[2] S. Bechhofer, D. De Roure, M. Gamble, et al. Research objects: Towards exchange and reuse of digital knowledge. 2010.

[3] A. BÃąnÃąti, P. Kacsuk, and M. Kozlovszky. Classification of scientific workflows based on reproducibility analysis. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 327–331, May 2016.

[4] E. Deelman, J. Blythe, Y. Gil, et al. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*, pages 11–20. Springer, 2004.

[5] Y. Gil, E. Deelman, M. Ellisman, et al. Examining the challenges of scientific workflows. *Ieee computer*, 40(12):26–34, 2007.

[6] C. A. Goble, J. Bhagat, S. Aleksejevs, et al. myExperiment: a repository and social network for the sharing of bioinformatics workflows. *Nucleic acids research*, 38(suppl 2):W677–W682, 2010.

[7] P. J. Guo and D. R. Engler. CDE: Using System Call Interposition to Automatically Create Portable Software Packages. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, pages 21–21, Berkeley, CA, USA, 2011. USENIX Association.

[8] K. M. Hettne, K. Wolstencroft, K. Belhajjame, et al. Best Practices for Workflow Design: How to Prevent Workflow Decay. In *SWAT4LS*, 2012.

[9] B. Howe. Virtual Appliances, Cloud Computing, and Reproducible Research. *Computing in Science Engineering*, 14(4):36–41, 2012.

[10] H. Meng, R. Kommineni, Q. Pham, et al. An invariant framework for conducting reproducible computational science. *Journal of Computational Science*, 9:137–142, 2015.

[11] H. Meng and D. Thain. Umbrella: A portable environment creator for reproducible computing on clusters, clouds, and grids. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '15, pages 23–30, New York, NY, USA, 2015.

[12] H. Meng, A. Vyushkov, M. Wolf, et al. Conducting Reproducible Research with Umbrella: Tracking, Creating, and Preserving Execution Environments. In *e-Science (e-Science), 2016 IEEE 12th International Conference on*, 2016.

[13] H. Meng, M. Wolf, P. Ivie, et al. A case study in preserving a high energy physics application with Parrot. volume 664, page 032022. IOP Publishing, 2015.

[14] D. Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 2014(239):2, 2014.

[15] T. Oinn, M. Addis, J. Ferris, et al. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[16] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields. *Workflows for e-Science: Scientific Workflows for Grids*. Springer Publishing Company, Incorporated, 2014.

[17] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency and computation: practice and experience*, 17(2-4):323–356, 2005.

[18] J. Yu and R. Buyya. A Taxonomy of Scientific Workflow Systems for Grid Computing. *SIGMOD Rec.*, 34(3):44–49, Sept. 2005.

[19] J. Zhao, J. M. Gomez-Perez, K. Belhajjame, et al. Why workflows break - Understanding and combating decay in Taverna workflows. In *E-Science (e-Science), 2012 IEEE 8th International Conference on*, pages 1–9. IEEE, 2012.

[20] C. Zheng and D. Thain. Integrating Containers into Workflows: A Case Study Using Makeflow, Work Queue, and Docker. In *Proceedings of the 8th International Workshop on Virtualization Technologies in Distributed Computing*, VTDC '15, pages 31–38, New York, NY, USA, 2015.