Vitor Carvalho Neves

MANAGING IMPLICIT DATA PROVENANCE IN SCIENTIFIC EXPERIMENTS

Thesis presented to the Computing Graduate program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic area: Software Engineering.

Advisors: Prof. D.Sc. Leonardo Gresta Paulino Murta
Prof. D.Sc. Vanessa Braganholo Murta

Niterói
2014

VITOR CARVALHO NEVES

MANAGING IMPLICIT DATA PROVENANCE IN SCIENTIFIC EXPERIMENTS

Thesis presented to the Computing Graduate program of the Universidade Federal Fluminense in partial fulfillment of the requirements for the degree of Master of Science. Topic area: Software Engineering.

Approved on September 2014.

APPROVED BY

Prof. D.Sc. Leonardo Gresta Paulino Murta – Advisor

IC-UFF

Prof. D.Sc. Vanessa Braganholo Murta – Co-Advisor

IC-UFF

Prof. D.Sc. Daniel Cardoso Moraes de Oliveira

IC-UFF

Prof. D.Sc. Marta Lima de Queirós Mattoso

UFRJ

Niterói

2014

I dedicate this work to God, for everything that I am and have. Nothing happened or will happen in my life without His permission. Without merit great things already done for me. I will always praise Him.

ACKNOWLEDGMENTS

I would like to thank God for the opportunity to be engaged on this work and for everything in my life.

I would like to thank my parents and brother for supporting my decisions.

I would like to thank Fernanda Freire de Araújo for all support and patience along this work. Without you by my side, it would have been much harder to walk this path. Thank you.

I am grateful to my father and uncles by always stimulating my curiosity and wish to learn and improve my knowledge. This defines much of who I am. In special, José Ataide da Silva and Leir da Silva Carvalho guided me on my first steps on the computing science.

I want to thank Professor Anselmo Antunes Montenegro by the encouragement to get involved on this course. Much of this result is product of our talking and your encouragement and counsels. Thank you.

This thesis would not have been possible without the help, patience, and counsels of my advisors, Leonardo Murta and Vanessa Braganholo. Thank you.

I want to thank Kary Ocaña and Professor Daniel Oliveira by counsels, support, and for providing the environment to the evaluation of this work.

I want to thank my fellow postgraduate students in the computing science department and in GEMS for promoting a stimulating and welcoming academic and social environment. In special to my friends, Marcos Vinicius Policarpo Cortês and Daniel Heráclio Olsen Maia do Carmo by all talk, counsels and support along this work.

I want to thank my job fellows for all the support along the development of this work. Sometimes it is hard to conciliate both worlds, and it would not have been possible without your comprehension and support. Special thanks to Carlos Eduardo Reis Ferreira, Sebastian Höfle, Tácio Diogo, Ana Cristina Florentino Ferreira, Adriano Simões and Fernando Mezini.

I would like to acknowledge the financial, academic, and technical support of the Universidade Federal Fluminense and FAPERJ.

"Tighter integration between scientific workflows and file management is necessary to enable the systematic maintenance of data provenance."

(KOOP *et al.*, 2010a)

RESUMO

Experimentos científicos representados como workflows científicos podem criar, alterar ou acessar dados não explicitamente referenciados na especificação do workflow, levando a fluxos de dados implícitos. A falta de conhecimento a respeito de fluxos de dados implícitos dificulta o entendimento e a reprodutibilidade de experimentos. Nesse trabalho, apresentamos o ProvMonitor, uma abordagem que identifica a criação, alteração e acessos aos dados mesmo em fluxos de dados implícitos. Além disso, ProvMonitor relaciona as informações capturadas com a atividade do *workflow* que as produziu, permitindo aos cientistas comparar os dados produzidos durante uma execução ou entre diferentes execuções do mesmo *workflow*. Essa comparação permite identificar efeitos colaterais na evolução dos dados provocados por fluxos de dados implícitos. O ProvMonitor foi avaliado e foi possível identificar sua capacidade de responder a consultas de proveniência em cenários que demandam conhecimentos específicos a respeito da proveniência de fluxos de dados implícitos.

Palavras-chave: workflows científicos, sistemas gerenciadores de workflows científicos, proveniência, fluxo implícito de dados, proveniência retrospectiva, proveniência implícita, gerência de configuração, sistemas de controle de versão.

ABSTRACT

Scientific experiments represented as scientific workflows may create, change, or access data products not explicitly referenced in the workflow specification, leading to implicit data flows. The lack of knowledge about implicit data flows makes it hard to understand and reproduce the experiments. In this work, we present ProvMonitor, an approach that identifies the creation, changing, or access of data products even within implicit data flows. Additionally, ProvMonitor links this information with the workflow activity that generated it, allowing scientists to compare data products within and throughout trials of the same workflow, identifying side effects on data evolution caused by implicit data flows. We evaluated ProvMonitor and observed that it is able to answer queries for scenarios that demand specific knowledge related to implicit provenance.

Keywords: scientific workflow, scientific workflow management systems, provenance, implicit data flow, retrospective provenance, implicit provenance, configuration management, version control systems.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS AND ABBREVIATIONS

SWfMS – Scientific Workflow Management Systems
CM – Configuration Management
VCS – Version control system
PGA – Provenance-gathering activities
OPM – Open Provenance Model
CLI – Command-line interface
API – Application programming interface
JVM – Java Virtual Machine
DAO – Database Access Objects
GUI – Graphic user interface
MFS – Multi-fasta files
MSA – Alignment files
PH – Phylip files
PC1 – First provenance challenge
PC2 – Second provenance challenge
GUI – Graphical User Interface

TABLE OF CONTENTS

CHAPTER 1 – INTRODUCTION

1.1 CONTEXT

Scientific experiments based on simulations usually consume and produce large amounts of data (HEY; TOLLE, 2009). In these experiments, scientists may use different programs to perform specific activities. Data produced by one activity may be the input to other activities, thus creating data flows. This chain of activities that composes a scientific experiment is usually modeled as a scientific workflow (DEELMAN *et al.*, 2009; MATTOSO *et al.*, 2010). Complex engines called Scientific Workflow Management Systems (SWfMS) (DEELMAN *et al.*, 2009) manage scientific workflows.

In this context, data provenance helps scientists to answer queries related to experiment data transformation (*e.g.*, "How were data generated or changed?"). Just as on art artifacts provenance, data provenance is the historical information about data ownership and transformations. It is the metadata associated to the workflow specification and the workflow executions, such as activity configurations, parameter values, as well as consumed and produced data products. Provenance data helps scientists to confirm or refute their scientific hypothesis associated to their workflow executions (FREIRE *et al.*, 2008).

Another important use of provenance is reproducibility, which is one of the key aspects related to scientific experiments modeled as scientific workflows (DA CRUZ *et al.*, 2011). An experiment is "scientific" only if it can be reproducible. To reproduce a scientific experiment, scientists analyze provenance information. This reinforces the usefulness of provenance information on scientific experiments.

There are two forms of provenance in the context of scientific experiments (FREIRE *et al.*, 2008): prospective and retrospective provenance. Prospective provenance refers to the workflow specification. It identifies the activities (*i.e.*, steps or tasks) to be executed to generate the expected results. On the other hand, retrospective provenance refers to the workflow execution (*i.e.*, trial). It captures executed activities and may include information about the environment used for data transformation. It works as an experiment execution log enriched by semantic information.

There are three main strategies for provenance gathering that could be used to classify provenance-gathering mechanisms (FREIRE *et al.*, 2008). The first (workflow-based) is to inspect the workflow during each trial to register produced and consumed data. The second (operation system-based) is to inspect the operating system to capture all system calls during a

trial (not directly related with the workflow notation). Finally, the third (activity-based) is through instrumentation of each workflow activity to correlate produced data with the activity responsible for the data production. Workflow-based mechanisms are the built-in infrastructure for provenance gathering in SWfMS. In this case, the SWfMS becomes responsible for gathering provenance information. By knowing the workflow specification and controlling its execution, these approaches are capable of capturing both prospective and retrospective provenance. However, they are SWfMS dependent. Then, it is difficult to integrate provenance collected from different workflows (in different SWfMS) that compose the same experiment. Moreover, these approaches can only capture provenance related to data that were explicitly specified in the workflow, thus missing provenance of implicit data flows.

Operating system-based (OS) mechanisms rely only on the OS environment's ability to capture data and dependencies among processes and data. These approaches are SWfMS independent, do not require any workflow adaptation, and are able to capture provenance even when implicit data flows are in place. However, they only capture retrospective provenance. Gathering provenance trough OS leads to fine-grained information about all system calls and files touched during the trial. The large amount of retrospective data can make these approaches prohibitive when scientists want to understand the experiment trial in terms of the activities specified in the workflow (*i.e.*, prospective provenance).

Finally, activity-based mechanisms adapt workflow activities, making them able to gather their own provenance. These approaches are capable of gathering both prospective and retrospective provenance and can be SWfMS independent. However, some activities are not easily adaptable, so they are wrapped as black boxes. In such situation, the content of an activity and its input and output data may not be explicitly specified, also leading to implicit data flows misses during provenance gathering. Additionally, this mechanism suffers from activity's instrumentation difficulties and overhead. It is possible to minimize the instrumentation overhead with the adoption of mechanisms capable of automatic instrumentations of the workflow activities, such as ProvManager (MARINHO *et al.*, 2011b).

1.2 BACKGROUND: ACTIVITY INSTRUMENTATION WITH PROVMANAGER

ProvManager (MARINHO *et al.*, 2011b) is an approach that operates by adapting workflow activities, allowing the adapted activities to gather provenance by themselves during a workflow trial. It minimizes the overhead of activities instrumentation via an automatic adaptation process. Additionally, by gathering provenance with an activity-based strategy, ProvManager can gather provenance even on workflows that use different SWfMS at

the same time, thus promoting the provenance gathering from workflow level to the experiment level (workflow compositions).

To be able to automatically instrument workflows activities, ProvManager relies on instrumentation plug-ins. Each plug-in is able to handle with a different SWfMS. Then, through new plug-ins, it is possible to extend ProvManager's SWfMS support.

During the workflow instrumentation, ProvManager collects prospective provenance and automatically wraps the workflow activities into composite activities. These composite activities consist on the original activity and some provenance-gathering activities (PGA). The PGA are responsible to gather retrospective provenance during the workflow trial. By doing this, the workflow maintains its original specification appearance (*i.e.*, data dependencies amongst activities and number of parameters), although with provenance-gathering capabilities.

It works as follows: the scientist uploads the workflow specification to the ProvManager. The prospective provenance is gathered while workflow activities are automatically wrapped with PGA. Then, scientists download the instrumented workflow specification, which is used to execute the experiment. During the trial, the wrapped activities gather the retrospective provenance through the PGA and send it to the ProvManager central provenance repository.

Although ProvManager minimizes the overhead of activity instrumentation (through automatic instrumentation), it works only with is specified in the workflow specification. Thus, it is not capable of gathering provenance of implicit data flows. Actually, even the content of files referenced in the workflow specification is missed.

1.3 MOTIVATION: IMPLICIT PROVENANCE

Despite the best efforts of scientists to specify all the experiment details (*i.e.*, activities, data files, data dependencies, etc.) into the workflow, sometimes the workflow specification is not complete in terms of consumed and produced data. For example, the workflow specification may reference a directory, but may not specify which data files in the directory are used. In such situations, some activities can read and write data that are not explicitly mentioned in the workflow specification, thus leading to implicit data flows within the specified workflow (MARINHO *et al.*, 2011a). Provenance related to this implicit data flow, which is ignored by all SWfMS (ALTINTAS; BARNEY; JAEGER-FRANK, 2006; FREIRE *et al.*, 2008; KOOP *et al.*, 2010b; MARINHO *et al.*, 2011a, 2011b), is called ***implicit provenance*** (MARINHO *et al.*, 2011a). Implicit provenance can be useful to help scientists to

identify and understand the influence of these implicit data flows. Since they occur implicitly and can influence experiment results, their effects commonly remain hidden from scientists, leading to misleading analysis. OS-based provenance gathering approaches (DAVISON, 2012; FREW; METZGER; SLAUGHTER, 2008; MUNISWAMY-REDDY *et al.*, 2006) manage to capture part of this provenance. However, they do not relate prospective and retrospective provenance.

This is even worse when the same data product (*e.g.*, file) is overwritten by several activities during a trial, hiding all traces of temporary contents consumed by some activities. Indeed, file overwrites is another problem that can lead to misleading analysis if not adequately treated even on explicit data flows (we show a real example of a workflow that is impacted by this problem in Chapter 3). This problem becomes harder in workflows that run in parallel, where parameter sweeps (WALKER; GUIANG, 2007) are performed and several workflow trials take place concurrently. On parameter sweeps, the same workflow is executed repeatedly with minor changes in its parameters, and provenance helps to explain the obtained results. Thus, this scenario imposes an additional requirement: versioning of intermediate data products that belong to explicit and implicit data flows.

Let us illustrate the implicit data flow scenario using two real workflow examples from different research groups. The first one is a VisTrails workflow that was downloaded from Crowdlabs (id 101)[1] and aims at visualizing salinity from Columbia River. In one activity, the scientists specified a *Pythonsource* module (which allows scientists to write their own Python code) that receives a data file path as input, generates a different data file, and sends its path as output. One problem here is that other data files can be produced or modified within the *Pythonsource*. In such cases, VisTrails is unaware of this information. Another example is a VisTrails bioinformatics workflow provided by Cruz et al. (2008). In this workflow there is one activity named *DatFrag* that divides the input data file into small fragments that are processed in parallel. However, only the output directory is informed, thus VisTrails is not aware of the data products generated by this activity. Although *Pythonsource* modules in VisTrails improve flexibility in the workflow specification, they can influence in the provenance management since they may hide implicit provenance. Note that this is a problem, since their use is quite common: 79 of the 201 (39.3%) workflows published in Crowdlabs use *Pythonsource* modules.

[1] http://www.crowdlabs.org/vistrails/workflows/details/101/

To illustrate some "incomplete" specifications that may affect provenance management, Figure 1 presents a synthetic workflow, intentionally simple for didactic reasons. The *UnzipFile* activity sends data to the *ApplyFilter* activity through a shared file (Img.bmp). The workflow specification, however, does not register this information (this is hard-coded into the activities). Activities *ApplyFilter* and *CutInterestingArea* also exchange information through the same file. Again, the workflow specification does not register this information. Instead, it only specifies that activity *ApplyFilter* sends a pair of parameter values (*i.e.*, filtervalues = {"circle", 2}) to activity *CutInteresingArea*. In this case, another problem arises: the consecutive changes over the same file may overwrite temporary data generated by the previously executed activities. Finally, activities *CutInteresingArea* and *IdentifyPhenomenon* illustrate the ideal scenario for provenance management, since the file they use is explicitly defined in the workflow specification ("c:\wksp\ImgCut.bmp"). In this last case, some problems may still arise. Again, if scientists execute the same workflow many times and they do not change the data file name (or path) for each execution, all the data generated in previous executions will be lost or altered.

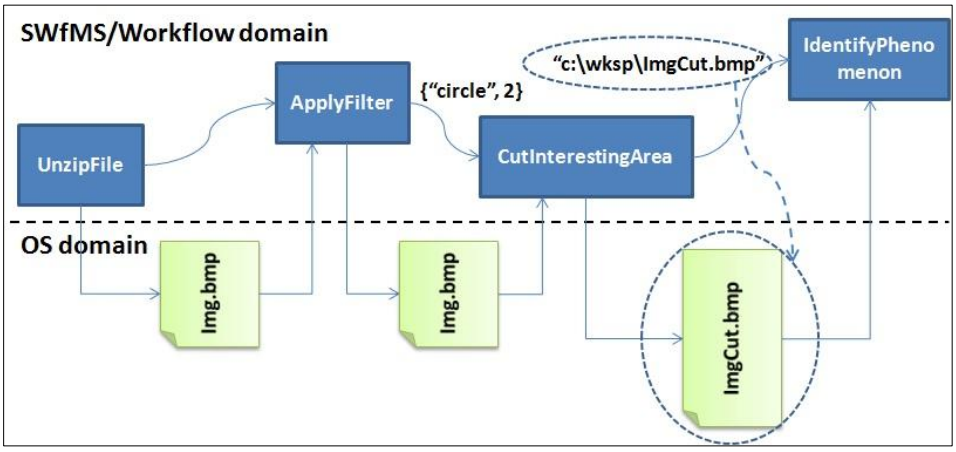


Figure 1: Implicit provenance example

Furthermore, Figure 1 illustrates that two data flows can take place in parallel when the workflow specification does not explicitly declare all manipulated data. One executes as specified in the SWfMS (*i.e.*, in the workflow domain) and presents data that are explicitly declared by the workflow and collected by the SWfMS. The other takes place out of the control of the SWfMS (*i.e.*, in the OS domain). It is an implicit data flow, as it is not declared in the workflow specification, but it still influences the results of the workflow. Since it occurs implicitly, its effects remain hidden from scientists. For example, when data files are overwritten during a workflow trial, all traces of temporary contents consumed by some activities are lost. Implicit provenance can be useful to help scientists identifying or

understanding this possible hidden influence of implicit data flows on workflow results. In addition, implicit provenance serves as an evidence of implicit dependencies amongst workflow activities. Identifying such dependencies can provide hints for experiment analysis.

In this work, we claim that it is fundamental to identify the relationship amongst workflow activities and the data products they implicitly changed because it is a key element to solve the implicit data flow problems. Thus, an implicit provenance gathering mechanism must rely on both workflow and OS domains. Actually, a tighter integration between scientific workflows and file management (OS domain) is necessary to enable the systematic maintenance of data provenance (KOOP *et al.*, 2010b).

1.4 GOALS

Given the aforementioned motivation, the aim of this work is to present an approach for gathering implicit provenance, relating it to the workflow activity that produced it, and being capable of gathering and versioning intermediate data files. The goal is to identify implicit data flows on files and gather provenance related to theses implicit data flows, allowing the scientist to identify the influence of such data flows on each activity execution and at the experiment results.

Thus, this work proposes a novel "hybrid" approach, which operates on both workflow and OS domains, for managing implicit provenance in scientific workflows through Configuration Management (CM). CM is a discipline used for controlling software evolution (DART, 1991). It is capable of identifying and registering changes on configuration items (*i.e.*, artifacts under CM), relating these changes with the issue that motivated them. By seeing experiment data as configuration items, it is possible to identify some similarities between provenance management and CM. Indeed, provenance is about identifying changes on experiment data (configuration items, in the CM terminology) that were motivated or generated by experiment activities (issues, in the CM terminology). Looking through this perspective, CM becomes a promising approach to provenance management.

The proposed approach, named ProvMonitor, goes in this direction, using the workflow specification as a source of issues to be tracked and employing a version control system (VCS) (ROCHKIND, 1975) and file system monitoring techniques to detect accesses (creations, changes, readings, and deletions) on files, relating them to the workflow activities that triggered these accesses. The use of a VCS allows ProvMonitor to file versioning and to gather the implicit provenance of the versions of files. Thus, ProvMonitor is able to distinguish the effects of each workflow activity over the data product versions. Thus,

ProvMonitor keeps a complete history of the data product changes associated to workflow activities executions.

1.5 RESEARCH QUESTIONS

The main objective of the proposed approach and its evaluation is to answer the following research question (RQ) and its secondary questions (SQ):

RQ: Does the implicit provenance gathering improve the perception about the impact of implicit data flow over activities executions and experiment results?

SQ: Does the implicit provenance gathering contribute to a more precise analysis about the experiment execution?

SQ: Does gathering of intermediate file content improve implicit provenance analysis?

SQ: Is it possible to make existing SWfMS aware of implicit provenance?

SQ: Does the overhead imposed by the implicit provenance gathering preclude its adoption?

1.6 CONTRIBUTIONS

The main contribution of this work is the gathering of implicit provenance with the intermediate files versions. Additionally, doing it through a CM perspective aware of the SWfMS behavior during the workflow trial opens some new interesting opportunities. It is a step forward compared to using CM tools only as storage of data or using it only to support the workflow development steps (such as versioning the workflow specification).

Therefore, this work presents a novel approach for managing implicit provenance in scientific workflows through CM. We discuss several ideas on how to use VCS to capture implicit provenance. Moreover, we present a thorough evaluation of ProvMonitor using a real large-scale bioinformatics workflow named SciPhy (OCAÑA *et al.*, 2011), which was modeled in SciCumulus (OLIVEIRA *et al.*, 2010) and executed in a public cloud. Results show that ProvMonitor can answer queries adapted from the Provenance Challenges, which demand specific knowledge related to implicit provenance.

Note that ProvMonitor is not a SWfMS by itself; it is a module for managing implicit provenance that can be coupled to existing SWfMS. ProvMonitor enriches the SWfMS's provenance repositories by adding information about implicit data flows. In fact, we have already coupled ProvMonitor with VisTrails (CALLAHAN *et al.*, 2006) and SciCumulus. Therefore, other workflow execution engines can also benefit from the proposed approach.

1.7 ORGANIZATION

This work is organized in five chapters. Chapter 2 presents the proposed approach named ProvMonitor. This chapter describes the main concepts behind the approach. It starts by describing the basic ideas of the approach and outlines an explanation about the instrumentation process, since ProvMonitor relies on such instrumentation concepts. After that, the general behavior of ProvMonitor gathering mechanism and analysis resources is presented, followed by a deeper discussion about the concepts that ProvMonitor relies on. Finally, the provenance model used and the architecture of the prototype are presented.

Chapter 3 presents a case study using a real workflow of the Bioinformatics domain. It describes the workflow, outlines the SWfMS used (SciCumulus), and describes the analysis performed to evaluate the approach. The evaluation aims at answering the research questions, evaluating the implicit provenance gathering capabilities of the proposed approach, the benefits of implicit provenance gathering and the overhead on implicit provenance gathering imposed in the workflow trial.

Chapter 4 outlines the related work. It presents existing approaches for provenance gathering that gather implicit provenance on some level. It also describes approaches that use VCS someway as support for provenance management (storage systems or to workflow specification versioning). Finally, it presents a comparison of the features available on each of the identified related work, regarding to implicit provenance gathering and VCS usage on provenance management, with the features of ProvMonitor.

Finally, Chapter 5 concludes this work. It describes the conclusions reached by the evaluation and discusses the contributions of this work. It also describes the limitations identified during the evaluation. Finally, it presents a set of opportunities for future work.

CHAPTER 2 – PROVMONITOR APPROACH

2.1 INTRODUCTION

The perception of implicit data flows relies on the knowledge of the explicit data flow. In fact, implicit provenance can be defined as the retrospective implicit data flow provenance related to its prospective provenance. Thus, an implicit provenance gathering mechanism must rely on both workflow and OS domains. According to Koop et al. (2010b), a tighter integration between scientific workflows and file management (OS domain) is necessary to enable the systematic maintenance of data provenance. Provenance maintenance entails avoiding provenance loss in multiple consecutive trials that overwrite files and preserving data generated during the workflow trial.

File management (or at least data management) is a target of different areas such as databases and CM. In the CM field, VCS are responsible for managing the different versions of configuration items (artifacts under management, *e.g.*, file), which is done by managing a workspace for the configuration items. According to Koop et al. (2010b) VCS capture the changes, but not why these changes occurred. Indeed, CM uses another system for storing such information: issue-tracking systems. Usually, each *issue* (the representation of a change specification, *e.g.*, a task) is associated with the *configuration item* versions produced during the *issue* fix. This integrated CM infrastructure can lead to a complete perception of when, how, where, what, why, and by who a configuration item version was created (DANTAS; MURTA; WERNER, 2007). In the context of workflow provenance data, the workflow specification defines the *issues* that motivate changing artifacts. Then, relating provenance data with the workflow activity that generated it is equivalent to relating a VCS *check-in* (VCS command that produces a *commit*, versioning the artifact) with its respective *issue* in the issue-tracking system.

It is interesting that, in practice, the workflow execution is somehow analogous to software development, if looked through the provenance gathering perspective, since in both cases there is a set of issues that motivates changes on data. On both cases, it is of interest to keep the trace of changes. This reinforces the benefits of relating prospective and retrospective provenance, since that relating prospective and retrospective provenance allows a complete perception of when, how, where, what, why, and by who (workflow step/activity) a change occurred during a workflow trial.

In software development, the challenge of CM is to guarantee that the software evolves in a controlled way. Thus, it is necessary to provide isolation for the developer when working on an issue to avoid interference with other developer (*e.g.*, changing the same file at the same time). However, after finishing the work on the issue, the changes made in parallel must be combined, allowing the inclusion of multiple issues in the next release.

The development process relies on VCS repositories that store the source code of the software being developed to share the same code base among multiple developers. However, the developers do not change files directly on the repository. Instead, in order to obtain isolation during the work on an *issue*, the developers copy the code base to their own workspace. The workspace is a directory in the developer computer. The workspace preparation involves retrieving files from an existing repository (*clone* command). After finishing the work on an issue, the developer must check-in the changed files (*commit* command), which gathers all changes in the workspace and generate a version id to identify the new versions of files. By operating in workspaces, different developers may work upon the same versions of files at the same time, without working over the same instance of the same files, avoiding conflicts that could lead to inconsistency. Finally, the developer may need to share the changes with other developers. This is done by pushing back the new versions of files to the central repository (*push* command).

Different developers may change the same version of the same files concurrently. In such situation, pushing back the changes to the repository forks the history of the files changes in different *branches*. It occurs because two different versions of files are created from a common ancestor instead of one being a direct evolution of the other. For some reasons, the history of changes may be explicitly branched (*branch command).* For example, if a subset of developers want to work on a set of coupled issues sharing code only between them. Thus, there are different types of branches. The implicit branches (created by *pushes* with concurrent changes over the same files versions) and the *named branches* (created explicitly through the *branch* command). The *named branch* is a branch explicitly created with a name as an annotation, facilitating references to the branch. Different *branches* may be combined to restore a single evolution line to the software development history (*merge command*). Finally, it is possible to mark a set of versions of files. It could be done using *tags.* A *tag* works as an annotation that identifies a specific configuration of files versions in the repository.

Some work (DAVISON, 2012; GUO; SELTZER, 2012) already used VCS to support the workflow development by versioning workflow specification and inputs/outputs of each

version, or even for storing files. However, as this work focuses on the implicit provenance gathering (*i.e.*, implicit data flows provenance) the focus is on the trial (execution) instead of other moments during a workflow life cycle (*e.g.*, composition and analysis) (MATTOSO *et al.*, 2010).

Therefore, there are four main steps for provenance management through CM: identify the issues to be tracked (prospective provenance register), generate an experiment execution identifier (*i.e.*, *trial id*), prepare the workspace for the execution, and gather the retrospective provenance, relating it to corresponding prospective provenance. First, prospective provenance must be gathered. Once identified, workflow activities can be treated as issues that motivated the changes gathered as retrospective provenance. Then, a trial identifier must be generated to be possible to identify the tuple {*trial id*, *activity id*} that produced the retrospective data during the trial. After that, the workspace must be prepared to allow retrospective provenance gathering on files. The workspace preparation involves creating a new workspace or retrieving files from an existing one (such as from a central repository through a *clone* command), and generating an annotation to identify the trial (through a *branch* command). Finally, at the end of each activity the new version of every changed artifact must be registered (through a *commit* command) and related to the tuple {*trial id*, *activity id*}. The use of the pointed commands (VCS commands) by the approach is better explained on Sections 2.3 and 2.5.

This vision of provenance management through CM motivated us to conceive a novel approach called ProvMonitor. The ProvMonitor approach was thought to be SWfMS-independent, and so it works by gathering provenance via an activity-based mechanism. However, to be capable of gathering implicit provenance, the gathering mechanism must also rely on the OS environment. Thus, ProvMonitor is a hybrid mechanism for provenance gathering, based on both activity and OS based mechanisms, capable of relating prospective and retrospective provenance.

It is important to highlight that an activity-based provenance gathering mechanism may impose work overhead to the scientist, caused by the activity instrumentation process. To minimize such effect, the ProvMonitor implementation was thought to be compatible with ProvManager (MARINHO *et al.*, 2011b), an activity-based provenance gathering mechanism that works through automatic workflow instrumentation. This way, each activity becomes responsible for gathering its own provenance and the automatic instrumentation reduces the overhead of activities adaptation imposed to the scientist.

In order to explain the provenance management with ProvMonitor it is possible to split provenance management description in three different moments: workflow instrumentation, provenance gathering, and provenance analysis. First, to execute ProvMonitor during the workflow trial the workflow must be instrumented with provenance activities that call ProvMonitor. Then, during the trial, the instrumented activities call to ProvMonitor allows ProvMonitor to gather retrospective provenance and relate to the "caller activity". Finally, it is possible to query the gathered provenance to analyses the experiment execution.

The remaining of this Chapter presents and explains the ideas of ProvMonitor with more details presenting an overview of the approach. To do so, it provides an explanation about workflow instrumentation on ProvMonitor. After that, the Chapter presents a discussion about provenance gathering and provenance analysis with ProvMonitor. Then, the Chapter follows by a deeper discussion about the concepts behind ProvMonitor (isolation strategies). Finally, the provenance model and the prototype architecture and implementation are presented.

2.2 WORKFLOW INSTRUMENTATION

As ProvMonitor is SWfMS-independent, it relies on instrumented workflow activities with specific provenance gathering mechanisms. To do so, the instrumentation step instruments the workflow injecting two "provenance activities" around the original activity, one before and another after the original activity. These "provenance activities" retrieve the files required by the activity, creating the workspace and committing the generated/modified files, respectively.

The idea of ProvManager integration is that during prospective provenance gathering, performed by ProvManager, the plug-in responsible for the instrumentation adapts the workflow with ProvMonitor's CM commands embedded in the PGA. This allows ProvMonitor to work during the workflow trial, gathering the retrospective provenance and associating it with the previously collected prospective provenance.

It is important to highlight that ProvManager alone is not capable of gathering implicit provenance. Actually, even the content of files referenced in the workflow specification is missed. ProvMonitor may be coupled to ProvManager as an extension of the ProvManager gathering mechanism that fixes such limitations. While fixing ProvManager limitations, ProvMonitor also benefits from the automatic adaptation process of ProvManager. Thus, our approach introduces implicit provenance aware PGA in ProvManager.

Figure 2 illustrates the instrumentation process for capturing implicit provenance in the theoretical workflow used as example in Chapter 1. Figure 2(A) shows the original workflow implemented in VisTrails. Figure 2(B) shows the instrumented workflow of Figure 2(A). It maintains the original appearance, except for two new activities: PGA0 and PGA1. PGA0 is responsible for preparing the ProvMonitor infrastructure to gather provenance. PGA1 is responsible for wrapping up the provenance data and pushing it back to a central repository. Finally, Figure 2(C) shows the details of the adapted activity *ApplyFilter*. The original activity together with three PGA composes the new (adapted) activity. PGA2 is responsible for preparing the workspace for the activity execution (this can vary depending on the used strategy - see Section 2.5). PGA3 is responsible for identifying and gathering changes and accesses to the workspace at the end of the activity execution. Finally, depending on which SWfMS or provenance mechanism is coupled with ProvMonitor, other PGA not specific for ProvMonitor may be employed to gather provenance data not related to file changes and accesses. For example, if coupled with ProvManager, PGA4 may be employed to gather activity parameters.

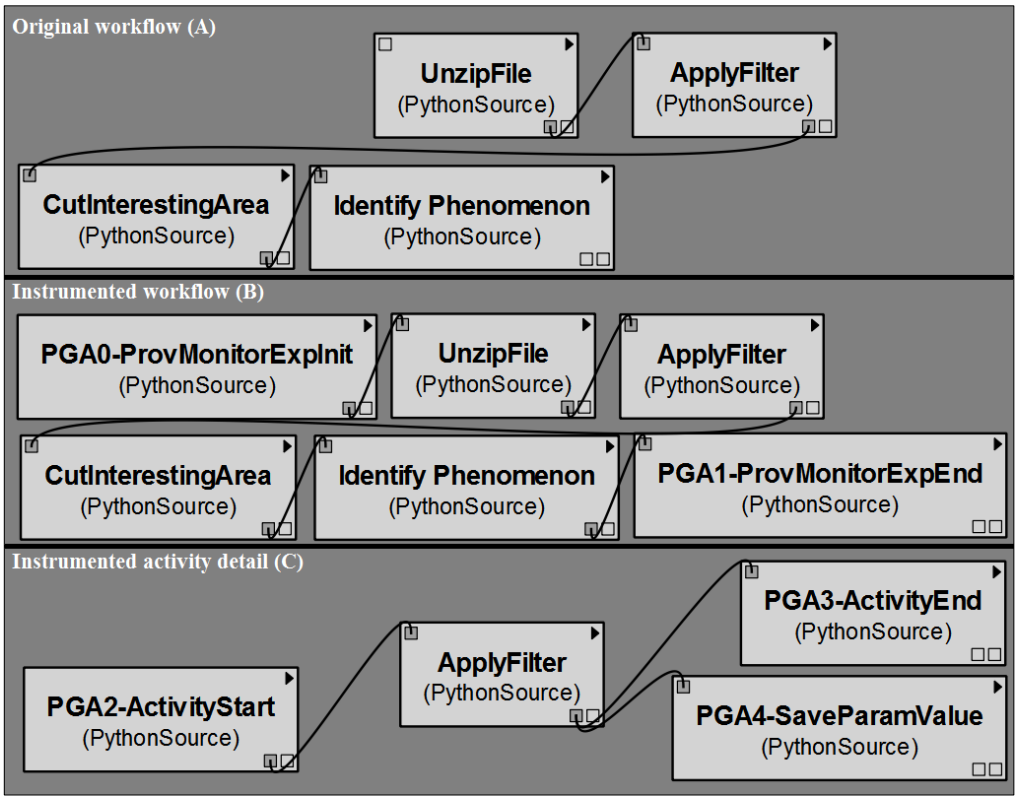


Figure 2: Original (A) and instrumented (B) workflow, and instrumented activity (C)

2.3 PROVENANCE GATHERING

ProvMonitor operates at runtime. Therefore, it assumes the prospective provenance has been already gathered, and the instrumented activities already points to an identifier of the instance of the respective activity. The gathering of prospective provenance occurs during the instrumentation process and can be done manually or automatically. The automatic capture and instrumentation may be done by ProvManager, other provenance mechanism or even by a SWfMS coupled with ProvMonitor. Whatever the process (manual or automatic) used for instrumentation and prospective provenance gathering, the only requirement is to provide an identifier for each instance of each activity inside the workflow. This is important to allow ProvMonitor to link the retrospective provenance with the prospective provenance already stored.

To operate on the OS domains, ProvMonitor relies on the file system, using two different resources. The first is files metadata. ProvMonitor verifies metadata to identify files access during the workflow trial. The second is a VCS. The VCS manages the workspace where the workflow executes, gathering all changes in files in the workspace during the trial. By managing the entire workspace, ProvMonitor is capable of gathering changes even when these changes are not explicit in the workflow specification, thus being able to capture implicit provenance. More details about the workspace manipulation are given on Section 2.5.

Gathering provenance at the OS domain may lead to a huge amount of provenance data. Additionally, the provenance data may include provenance not directly related to de workflow trial. Even when related to the trial, by relying only on the OS environment it is not possible, or at least not straightforward, to identify which workflow activity produced the effect gathered by OS domain provenance mechanism. Then, to be able to relate provenance gathered at the OS domain with the workflow activity that produced the effect observed in the OS domain, the provenance mechanism must also rely on the workflow domain, knowing the workflow activities being executed.

With this understanding, the ProvMonitor gathering mechanism works through activities instrumentation (detailed on Section 2.2). When an instrumented activity executes, it triggers the ProvMonitor OS mechanisms to gather provenance in the workspace. Doing so, ProvMonitor operates at the OS domains, but “surrounded” by workflows activities. Because it is triggered by an instrumented workflow activity, before and after the original activity execution, ProvMonitor is able to associate the provenance gathered in the workspace (OS domain) with the activity (workflow domain) that produced the changes in the workspace (the

activity that triggered the OS provenance mechanism). Operating at the OS domain surrounded by the workflows activities (workflow domain) has two benefits. The first one is the capability of associating prospective (workflow activity) and retrospective provenance (workspace changes). The second is to restrict the amount of provenance gathered at the OS domain to only provenance related to the workflow trial.

The gathering mechanism works by monitoring the experiment workspace, which is a directory in the computer where the workflow executes its activities. If necessary, one can use *chroot* command to set the workspace as the root directory, preventing accesses to other files in the computer. ProvMonitor uses VCS for managing the experiment workspace, capturing and storing file changes, including implicit provenance. All file changes or accesses are associated with the prospective provenance in the provenance database. To do so, file version ids (generated by the VCS) related to the version of each data file are associated with the activity that generated them. This allows for queries that relate both data and metadata. For example, one could ask which files were changed by a specific activity, or which activities accessed the files produced by other activities. File changes are gathered through VCS by comparing the state of a workspace on different moments in time. File accesses are identified by the comparison of access timestamps before and after the activities executions.

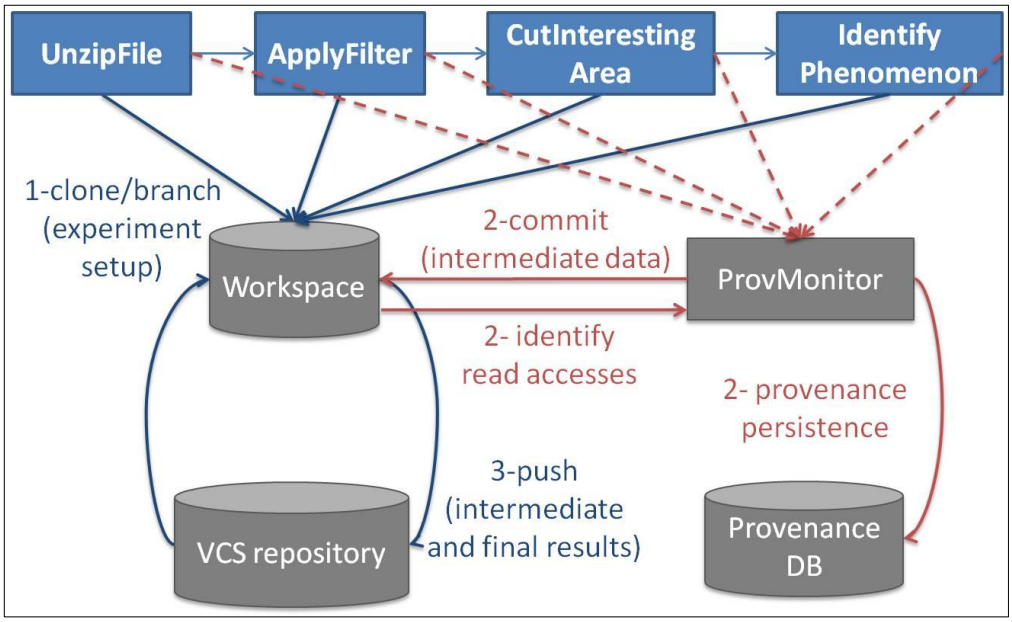


Figure 3: ProvMonitor provenance gathering overview

Figure 3 presents an overview of ProvMonitor gathering mechanism execution process that follows the steps described at Figure 4 during the execution of the sample workflow illustrated at Figure 1. As illustrated, at the beginning of the instrumented workflow trial (*Step 1 – clone* at Figure 3 and *Start ProvMonitor Infra* at Figure 4), ProvMonitor generates a trial

ID, prepares the workspace by cloning the necessary input files from a central repository (*VCS repository* at Figure 3) and generates a *named branch* for the current trial. Then, at the end of each activity execution (*Workflow Activity Execution* at Figure 4), ProvMonitor identifies the accessed files (*Identify Accessed Files* at Figure 4) and gathers all changes in the workspace. Then, ProvMonitor associates, in the provenance database, the accessed and changed files with the activity that triggered ProvMonitor (*Step 2 - commit* at Figure 3 and *Gather changes into workspace* at Figure 4). At the end, all changes in data files are pushed back to the central repository (*VCS repository*), making it available for further analysis (*Step 3 – push* at Figure 3 and *Push back to the Repository* at Figure 4). By managing the workspace, every change in the configuration items is caught by ProvMonitor, versioning every changed file after an activity execution, even if it is not explicitly defined in the workflow specification. It is important to highlight that ProvMonitor does all workspace manipulation automatically and transparently from the user. Therefore, scientists access provenance and the different versions of files content only at the repository, where all data of all trials are centralized.

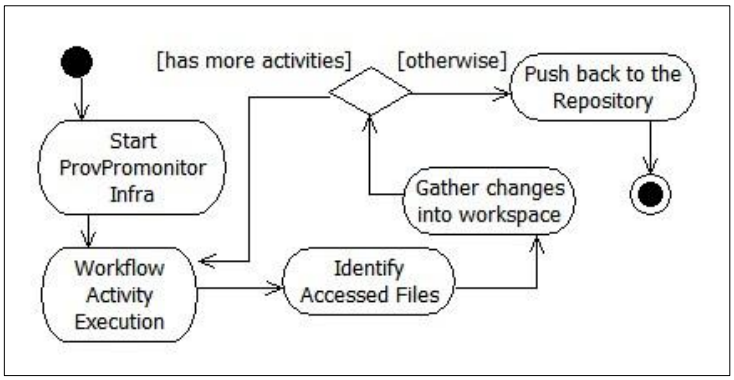


Figure 4: ProvMonitor gathering process

2.4 PROVENANCE ANALYSIS

A sample of an expected result of a workflow trial is shown at Figure 5. The workflow is shown at the top of the image. Below, the parallel lines connected to file names represent the history line of each file. The white and black dots represent access and changes to the files, respectively. It is important to note that every time a file is changed (black dot) a new version of it is created. For example, looking at the history line of file *Img.bmp* at trial 1 (lines above the dashed line), it is possible to identify that this file was changed by the activity *UnizipFile*, creating the first version of this file ($V1_{trial1}$). Then, activity *ApplyFilter* makes another change, creating the second version of the file ($V2_{trial1}$). Finally, activity *CutInterestingArea* reads the file, so ProvMonitor does not create a new version of it. As

previously discussed, this workflow has implicit data flows that could lead scientists to a misleading analysis. However, at the end of each activity execution, ProvMonitor catches all changes inside the workspace and identifies each file accessed during the activity execution.

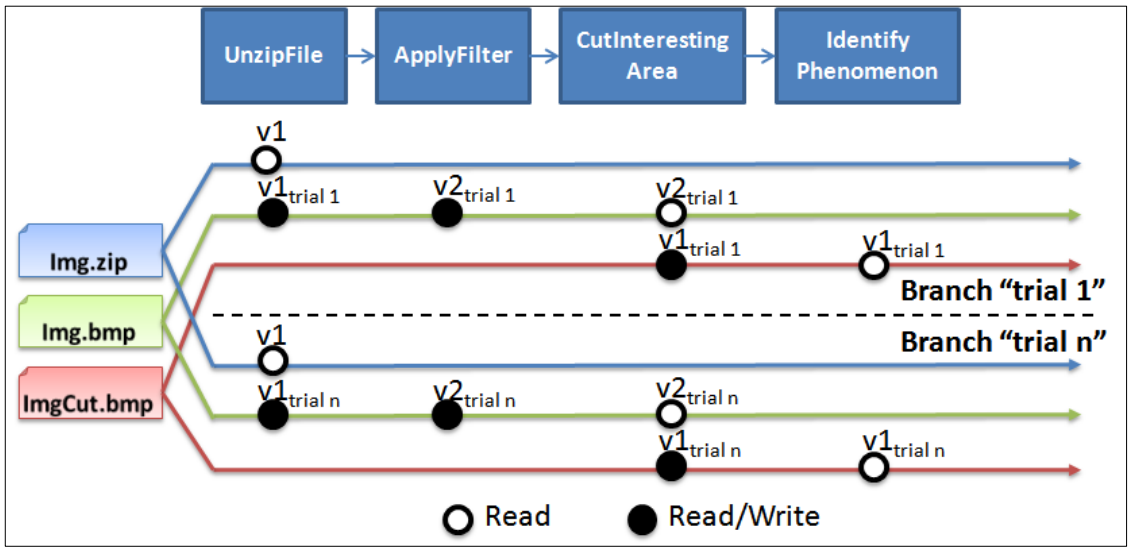


Figure 5: Activities executions changes through trials

Besides supporting coarse grain analysis, ProvMonitor also supports fine grain analysis considering the file contents. Therefore, ProvMonitor open opportunities for intra-trial and inter-trial analysis. The first one compares changes produced by two different activities at the same trial. The second compares changes produced by the same activity but in different trials. As we adopt VCS underneath ProvMonitor, it is possible to compute an on demand diff over any file version. For example, we can check exactly how an activity changed a specific file (*i.e.*, which lines were added and removed – intra-trial analysis). This opens opportunity to use syntactic diff (CONRADI; WESTFECHTEL, 1998) with specialized diff algorithms (COBENA; S. ABITEBOUL; MARIAN, 2002; HUNT; MCILROY, 1976; OHST; WELLE; KELTER, 2003; SILVA JUNIOR *et al.*, 2012) that are able to deal with different file types (XML, models, images, etc.). Moreover, we can contrast two trials and check how each file differs in these trials (inter-trial analysis). For instance, specialized diff algorithms could be used to compare $V1_{trial\ 1}$ and $V1_{trial\ n}$ of *ImgCut.bmp* file. Both versions were produced by the *CutInterestingArea* activity, but in different trials. Consequently, they were stored in different branches. This kind of analysis is useful to comprehend the effects of parameter sweeping on intermediate data.

2.5 ISOLATION STRATEGIES

The key perception behind ProvMonitor is the need for isolation in scientific workflows. We deal with three different levels of isolation inspired by CM: workflow, activity, and trial. At the workflow level, we allow the user to elect (during the instrumentation process) the experiment workspace. The workspace stores all files that are changed or accessed by the workflow, isolated from other files. ProvMonitor is able to register snapshots of such workspace in an efficient way through a VCS *commit* command, which computes the difference amongst the past and the current version of files in the workspace and stores this content difference in the repository. The sequence of snapshots constitutes retrospective provenance at file system level.

At the activity level, we isolate an activity from the remaining activities in the workflow trial. This isolation enables the identification of files produced by each activity. To do so, ProvMonitor adds a VCS *commit* command after the execution of each activity. This delimits a transaction per activity and registers all files changed or accessed within the transaction. ProvMonitor also can create a new isolated workspace for each activity execution. This new workspace is created using VCS *clone* command, which efficiently retrieves the workspace content. After the activity execution, the workspaces are shared through the VCS *push* command. Therefore, ProvMonitor allows activities to execute simultaneously, without one interfering with another.

At the execution level, we isolate different trials of the same workflow since there are several experiments that require multiple executions of the same workflow with different parameters values (*e.g.*, parameter sweep). ProvMonitor uses the VCS *branch* command to isolate one trial from the others. This way, each trial forks the repository history, creating a branch that evolves in a completely independent way of the remaining branches. Each branch has a name, thus allowing a precise identification of which trial produced which files. When necessary, branches can be combined back with VCS *merge* command. Branching can be used not only to isolate a trial from others, but also to isolate the history of concurrent activities in the same trial. Combining these histories (*i.e.*, merging the branches back) allows restoring the history line for the subsequent activity.

Table 1 summarizes the VCS resources used to achieve each isolation level. For workflow isolation, workspaces are used. For activity isolation, *commit* command, workspaces and branches are used. Finally, for trial isolation, ProvMonitor also uses branches.

Table 1: Isolation level versus VCS resource

Isolation level	VCS resource used
Workflow	Workspace
Activity	Check-in (Commit) / Workspace / Branches
Execution (trial)	Branches

The remaining of this section presents a deeper discussion about different strategies for using ProvMonitor. We discuss three different strategies used to provide isolation in ProvMonitor: workspace per workflow with a branch per trial, workspace per activity with a branch per trial, and workspace per activity with a branch per activity. These strategies tailor ProvMonitor to work with the idiosyncrasies of each SWfMS, and provide different effects in terms of provenance gathering and analysis. The strategies are presented from the simplest to the more complex one, regarding the complexity of VCS resources used.

2.5.1 WORKSPACE PER WORKFLOW WITH A BRANCH PER TRIAL

The first strategy creates a workspace for the entire workflow. After each activity execution, a commit is triggered at the workspace, identifying and gathering all file changes. This strategy is effective for a linear workflow, where all activities operate in sequence over the same workspace. Figure 6 shows the repository history after two successive trials. As it can be seen, branches are formed at the repository (blue and pink dots denote different branches). This occurs because each trial forks the history, using the same files versions as input ("*canonicalBranch*"), and has its own transformation sequence.

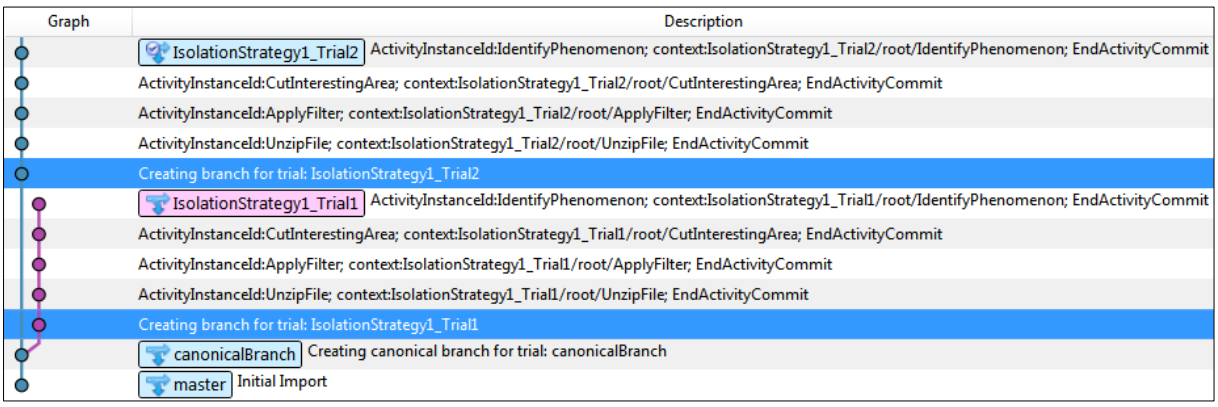


Figure 6: Workspace per workflow with a branch per trial (visualized using SourceTree[2])

[2] http://www.sourcetreeapp.com/

To mark the initial set of the workspace, a special branch named "*canonicalBranch*" is created. The "*canonicalBranch*" is used to facilitate restoring original files content. Thus, each new trial works upon the initial dataset instead of working on a dataset changed by the previous trials. As shown at Figure 6 the "*canonicalBranch*" is the fork point for each trial branch.

In fact, considering CM concepts, the "*canonicalBranch*" works more as tag than as branch, as it works as an annotation that identifies a specific configuration of files versions into the repository. Then, it is also possible to create tags for different set of files versions, for example, to the files versions after a trial (the results of the execution). Thus, the results of a trial may be used as input for subsequent trials, instead of using the original files as input for every trial. This could be useful on scenarios of iterative workflows, where the results of a trial are used as input of the subsequent trials.

ProvMonitor automatically names each trial branch with the trial ID, as shown in Figure 6 (see the highlighted names inside blue and pink boxes) to ease the identification of a trial branch. Through the named branches it is possible to identify the history of changes on a given trial isolated from other trials. This is useful for both analysis and storage, since named branches can be removed from the repository if desired.

2.5.2 WORKSPACE PER ACTIVITY WITH A BRANCH PER TRIAL

Some SWfMS may be incompatible with the previous strategy. For example, SciCumulus (OLIVEIRA *et al.*, 2010) was designed to execute in the cloud, allowing the execution of different activities in parallel in different virtual machines. Each parallel execution of an activity is called "cloud activity". Every time a cloud activity is about to start, SciCumulus creates a specific directory for it and the required files are copied to the new directory.

In such situations, the use of a single workspace (as a "root" of every cloud activities directories) for the entire trial may lead to data duplication inside the workspace. The same file may appear as two different files, each one of them on different paths. The files are treated as independent files and not considering that, one is a new version of the previous. Then, the file trace (history of file changes) is lost. The scenario described above was thought using workspaces on a shared area, for example, a shared storage. However, cloud based SWfMS or even distributed SWfMS may also uses different physical disks on different nodes on a cluster/grid to set the workspaces. Distributed configurations completely prevent the usage of a single workspace.

Thus, to work with SciCumulus, ProvMonitor has to create one workspace per activity, managing the workspace together with the SWfMS. The activities are isolated from each other in independent workspaces. ProvMonitor captures all file changes and accesses inside the workspace and relates them with the corresponding activity in the provenance database.

In this strategy, the workspace is cloned with all files produced by the previous activities, setting a managed workspace at the activities directory. At the end of each activity execution, the workspace is analyzed and all detected changes are *committed* and *pushed back* to the central repository, making them available to other activities. The use of this strategy on linear workflows generates the same results illustrated at Figure 6. It is important to notice that by using this strategy, the data linkage is not lost, since editions, moves, renames, and copies are done over the management of ProvMonitor.

2.5.3 WORKSPACE PER ACTIVITY WITH A BRANCH PER ACTIVITY

Using the strategy of a workspace per activity requires some special attention. When workflow activities are parallelized, they can make concurrent changes into the same file, leading to a file's history branching. Working on isolated workspaces, these changes occur at the working copies of the files, thus not influencing or interfering with the concurrent executions. However, at the end of the activities executions, the *commit* and *push back* propagates the branch history to the repository, leading to a scenario of workspace per activity with a branch per activity.

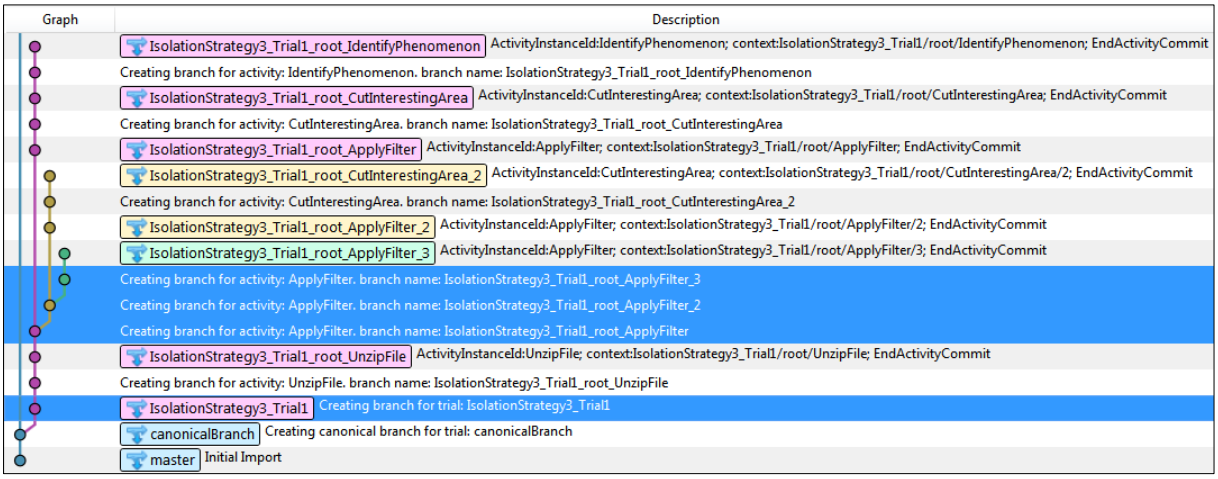


Figure 7: Workflow per activity with a branch per activity

Figure 7 illustrates how this history is stored in the repository. It is possible to identify that not every branch of the history continues. For example, three instances of the activity

ApplyFilter are executed concurrently (*ApplyFilter*, *ApplyFilter2* and *ApplyFilter3*) – the group of three lines highlighted in Figure 7. The figure shows that tree branches are created, representing the history of each one of the concurrent activities isolated.

Figure 8 presents a representation of these activities based on the provenance gathered and stored into the repository shown in Figure 7. The activities are represented by the rectangles and the arrows represent the data flow between the activities. It illustrates that only the first *ApplyFilter* has its execution line complete until the end of the workflow. The *ApplyFilter2* branch ends its evolution after the *CutInterestingArea2* execution, and the *ApplyFilter3* does not even start an instance of the *CutInterestingArea* activity.

This is interesting in several aspects. First, it helps to evidence activities that effectively contributed to the results of a trial. Second, it indicates the critical path of transformations, performed to the original files until their last version. Therefore, after the analysis process, if the scientist only desires to store the essential provenance information, it is easy to identify information that could be discarded.

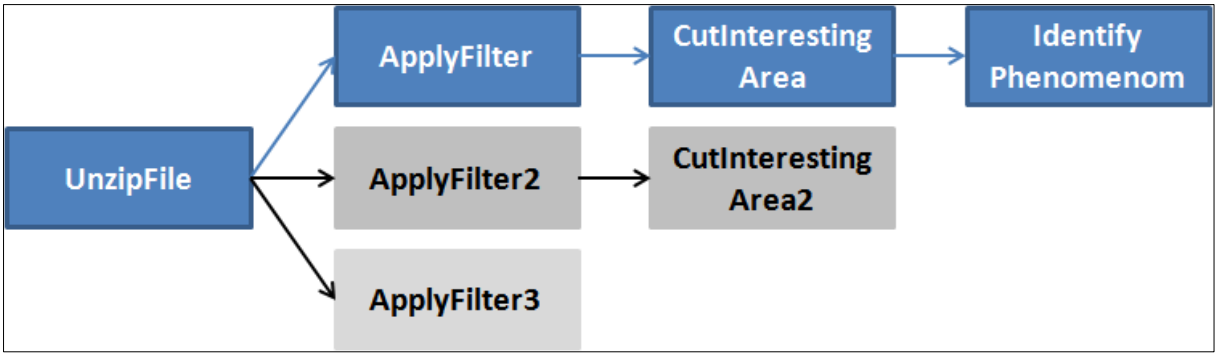


Figure 8: Workflow with concurrent activities execution sample

In the example illustrated at Figure 8, another aspect may be observed about workflows trials. During a trial, activities may produce a fork in the data flow, executing it concurrently or even sequentially. In the above situation, the forks, once created, flows until the end of the trial as independent branches. However, concurrently data flows may be combined to reintegrate a single data flow. This combination may lead to ***merge*** situations, which are discussed in the next section.

2.5.4 DATA FLOW MERGE ISSUE

Concurrent data flows may be combined, leading to a single data flow. This combination process may occur through the selection of one of the branches in detriment of others. A branch selection may produce a similar execution scenario as illustrated in Figure 8. However, instead of a branch selection, a complex combination of files from any of the

concurrent data flows may occur, for example, on MapRreduce workflows (DEAN; GHEMAWAT, 2008).

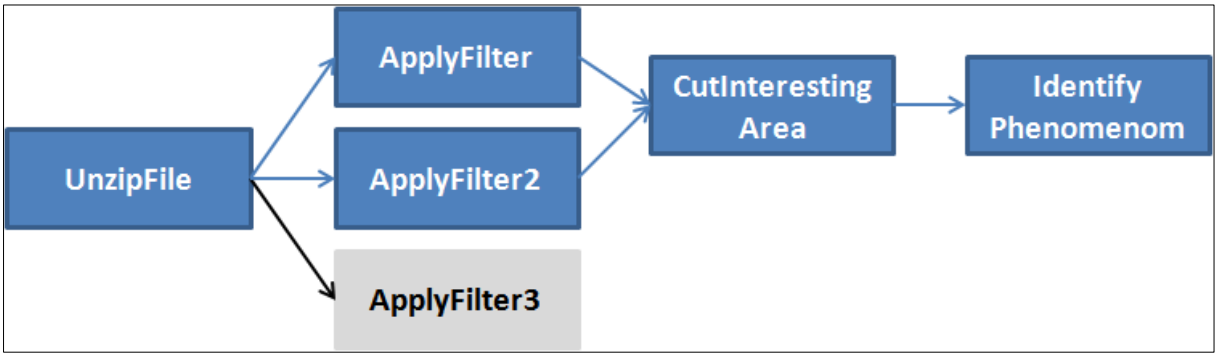


Figure 9: Workflow with concurrent data flows merge

Figure 9 illustrates a scenario of concurrent data flows merge. Both the activities instances *ApplyFilter* and *ApplyFilter2* contribute with inputs to the activity *CutInterestingArea*. Then, the workspace used as input to the *CutInterestingArea* has to be a merge of the outputs of the both instances *ApplyFilter* and *ApplyFilter2*.

The merge of concurrent data flows may lead to complex situations. For example, *ApplyFilter* and *ApplyFilter2* may produce, as output, files with the same names. How to decide what files must be used? It could be the joining of all files outputted by all data flows. Files of the first execution to finish may be used or the last execution to finish may overwrite the files of the first one. It could be a combination of files of both data flows (merge) with a file from each data flow. Alternatively, it could be a combination of each single file of each one of the data flows (merging each one of the files). In fact, the answer to this question relies on the semantic of the activities involved. Therefore, to automatic merge concurrent data flows, the semantics of the workflow has to be considered. Some SWfMS may provide special control flow activities that can help to solve this issue, such as decision points or synchronization points.

Despite the relevance of this issue, more elaborate merge strategies present complexities that need to be more carefully examined and are out of the scope of this work, opening opportunities for future work.

2.6 PROVENANCE MODEL

ProvMonitor was thought to be compatible with ProvManager, being coupled as an extension of ProvManager gathering mechanism to gather implicit provenance and files content. Thus, ProvMonitor inherits the ProvManager provenance model. The ProvManager provenance model is based on the Open Provenance Model (OPM) (MOREAU *et al.*, 2008).

However, it extends OPM to support prospective provenance. Despite the fact that ProvMonitor inherits ProvManager OPM-based provenance model, it is also possible to map the OPM-based provenance model to a PROV-based (LUC MOREAU; PAOLO MISSIER, 2012) provenance model (BIVAR *et al.*, 2013).

As ProvMonitor operates at runtime, it assumes the prospective provenance has been already gathered. Thus, the prospective provenance model used must be the prospective provenance model of the mechanism used on the automatic process of instrumentation and prospective provenance gathering. For the manual process, the scientist may use the model more suited for its needs. Whatever the process (manual or automatic) used for instrumentation and prospective provenance gathering, the only requirement is that the prospective model provides an identifier for each instance of each activity inside the workflow. This is important to allow ProvMonitor to associate the retrospective provenance with the prospective provenance already stored. In this work, we considered the ProvManager model to cover the prospective provenance.

ProvMonitor extends ProvManager provenance model (MARINHO *et al.*, 2011b) to capture implicit provenance in files and file contents. Our extension adds two new entities: Element Execution File Access and Element Execution Commit. Additionally, ProvMonitor uses the Element Execution entity from the ProvManager provenance model.

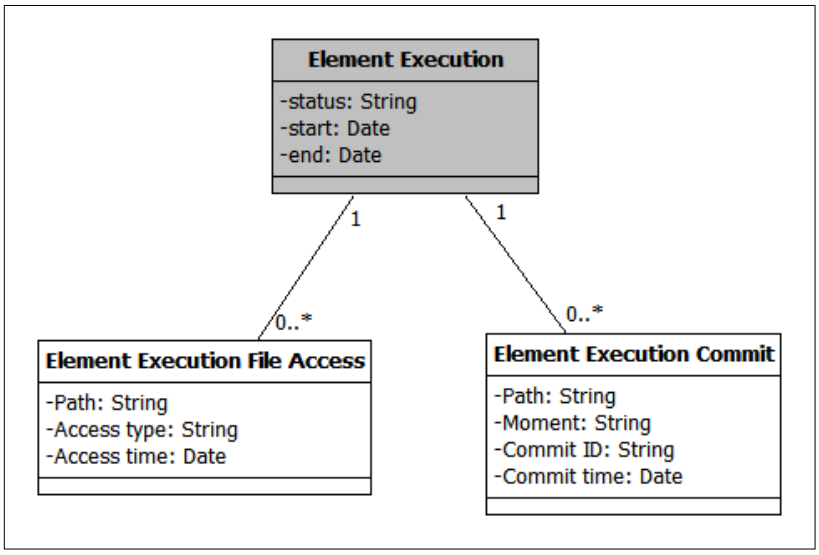


Figure 10: ProvMonitor provenance model

Figure 10 illustrates ProvMonitor provenance model, represented in a UML class diagram (OMG 2010). The ProvManager inherited entity is marked as a filled class (grey). The ProvMonitor entities are the white ones. The descriptions of these entities are as follows.

- The Element Execution entity stores information about a workflow element (*i.e.*, activity) execution on a given trial of the experiment;
- The Element Execution Commit entity stores all commit hashes (*Commit ID attribute*) triggered by each activity, registering the moment the commit was triggered, *e.g.*, start of the activity or end of the activity;
- The Element Execution File Access entity stores the access type *(create, change,* delete, or read) and the accessed files of each commit, which is related to each activity execution in the workflow.

Although the VCS repository stores provenance data in an unstructured way, with this extension of the provenance model, provenance information gathered by ProvMonitor is stored together on the same provenance database. Thus, scientists only need to access the VCS repository when they need to analyze file content. All other provenance queries can be executed directly over the provenance database.

2.7 IMPLEMENTATION

The prototype implementation of ProvMonitor uses Java. The choice of Java was for portability issues. Making ProvMonitor portable along different OS makes it easier to couple it to different environments. The VCS used was Git (CHACON, 2009). The choice of Git was because it is a distributed VCS with growing use on open-source projects and because it has references available to support development. Additionally, there are some interesting features supported by Git that could be used during the experiments. These features are related to performance optimizations during *commits* and repository storage optimizations, such as repository compression and versioning of file contents separated from file paths (two files with the same content but on different paths points to the same content object into the repository), reducing the amount of disk space used.

ProvMonitor provides a command-line interface (CLI) to facilitate interoperability with other systems, allowing the coupling of ProvMonitor as an external application. Thus, there is no need to change the SWfMS or the provenance mechanism when using ProvMonitor. However, ProvMonitor may also be included as a library (.jar) on Java applications, providing methods to the SWfMS to invoke ProvMonitor execution during the workflow trial.

Figure 11 presents the prototype architecture, with the main elements separated on layers related to their role into the architecture. The definitions of these layers are as follows.

- **Interface:** this layer represents the interface for interoperability with ProvMonitor. It presents a CLI, which allows SWfMS or provenance systems to invoke ProvMonitor as an external application. The methods supported by ProvMonitor are prepare to start an experiment execution; notify the end of an experiment execution; prepare to the start of an activity execution; and notify the end of an activity execution. It also presents an application programming interface (API) to be possible to couple ProvMonitor as a library (*e.g.*, including a .jar of ProvMonitor on Java applications) on other systems (Provenance Systems or SWfMS). When coupled as a library, the *Retrospective Business Services* provides the interface for invoking the provenance gathering methods supported by ProvMonitor
- **Control:** this layer represents the controller for the gathering methods supported by ProvMonitor. The *Retrospective Business Services* implements the supported gathering methods, and is responsible for orchestrating the execution of ProvMonitor's provenance gathering mechanism (Provenance gathering layer) and the communication between the provenance gathering layer and the persistence layer. It also provides the interface for invoking the provenance gathering methods from the API
- **Provenance gathering layer:** this layer represents the provenance gathering mechanisms used by ProvMonitor. Two interfaces compose this layer: a *Workspace Access Reader* and a *VCS Manager*. The *Workspace Access Reader* is responsible for reading the workspace files metadata in order to identify accesses to workspaces files. The *VCS Manager* is responsible for gathering changes (creation, removing, or changing of files) in the workspace and for storing all the gathered changes versioning the workspace. Both rely on the Operating System File system. This layer relies on interfaces to abstract the implementation of each one of the gathering mechanisms. Thus, it is possible to offer support for different VCS or File Systems. For instance, the current VCS implementation is Git and the implementation of the *Workspace Access Reader* relies on the Java 7 "*nio*" package that provides an abstraction of the file system metadata through Java Virtual Machine (JVM), improving portability.
- **Persistence: this layer rep**resents the persistence capabilities of ProvMonitor. The *Provenance model* is part of this layer. The persistence layer *relies on two*

i*nterfa*ces: *Provenance Database and the VCS Manager*. *T*he *Provenance Database* is responsible for the persistence of the provenance data gathered, according to the def*ined Provenance Model*. It may be accessed through Database Access Objects (DAO) interfaces, thus allowing abstracting the Database implementation. Indeed, ProvMonitor also allows the use of a web service (such as proposed by ProvManager) for provenance persistence instead of direct access to the database instance. The *VCS Manager* is responsible for storing the workspace file contents, and the different files versions generated during a trial. Through the *Provenance Model the inf*ormation stored by both interfa*ces* are related.

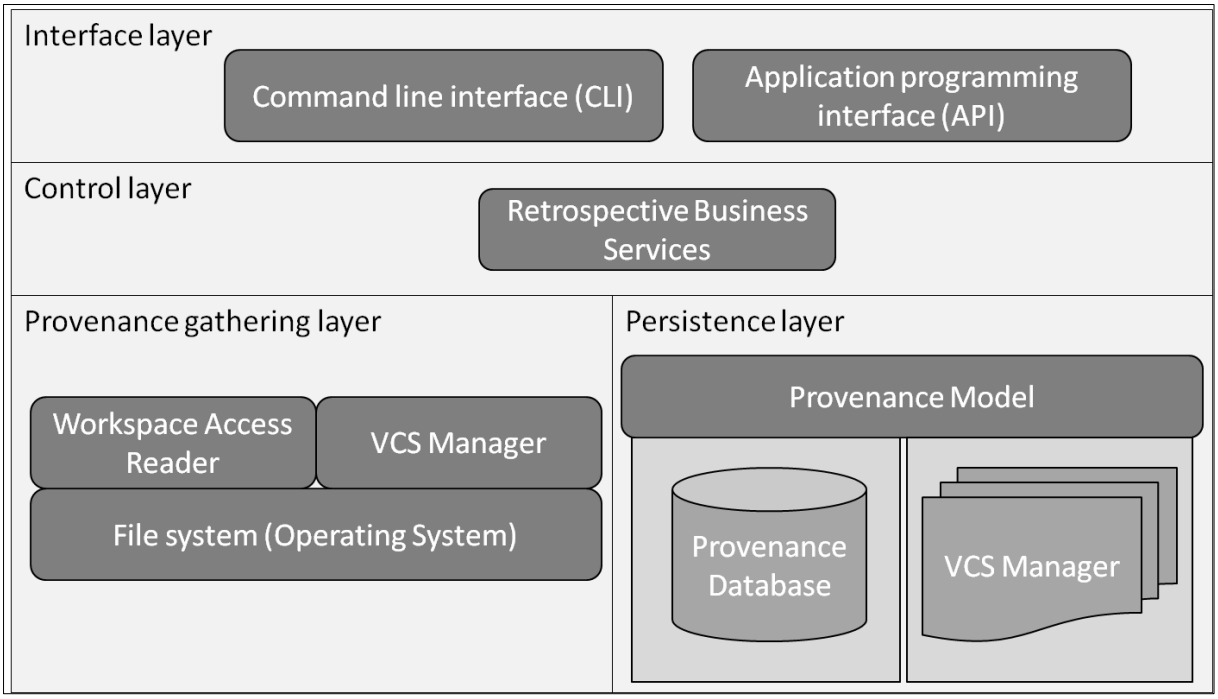


Figure 11: ProvMonitor architecture

Additionally, ProvMonitor has a property manager (working as a descriptor), which is not represented in the layers view (to simplify *the image), used* to parameterize some behaviors. The parameterized behaviors are the dat*abase used (e.g., M*ySQL, Pos*tegreSQL),* the da*tabase connection s*tring, the type of branching strategy (*e.g.*, branching per activity or branching per trial), t*he VCS implement*ation to be used (*e.g.*, Git, Hg), and the ProvMonitor internal log level (*e.g.*, Debug, Measure, Warning, Fatal, and others).

2.8 FINAL REMARKS

This chapter presented the ProvMonitor approach. The main objective of the approach is to gather implicit provenance while versioning intermediate data files. T*he proposal* is to

do it through a CM perspective. ProvMonitor is a step forward on the gathering of implicit provenance. It is also a step forward on the *usage of a CM pe*rspective on the provenance gathering, more specifically, to use VCS during a workflow trial, following the behavior of the trial, instead of using a VCS only as storage or to versioning workflow specifications.

The next chapter presents the evaluation of the proposed approach through a study case conducted with the objective of answering the research question and secondary questions of this work.

CHAPTER 3 – EVALUATION

3.1 INTRODUCTION

ProvMonitor was evaluated through a case study in the bioinformatics domain. The main objective was to find evidences of the benefits of gathering implicit provenance, to evaluate the proposed approach for gathering implicit provenance, and to evaluate how it could be useful for the scientist or someone analyzing or understating the execution of a scientific workflow. To do so, ProvMonitor was coupled with the SciCumulus (OLIVEIRA *et al.*, 2010) SWfMS to execute a scientific workflow related to phylogenetic analysis named SciPhy (OCAÑA *et al.*, 2011).

The choice of SciPhy was based on the availability of a specialist (pharmaceutical researcher) to assist on the evaluation about the semantic benefits of implicit provenance analysis. The choice of SciCumulus was also related to availability, since the current workflow implementation used by the specialist was built on SciCumulus. Additionally, coupling ProvMonitor with SciCumulus presented an opportunity to try a more complex and complete isolation strategy then using Vistrails. This also allowed us to evaluate the adaptability and generality of the concepts behind ProvMonitor.

To evaluate ProvMonitor we conducted both effectiveness and efficiency analysis. In the effectiveness evaluation, we analyzed if ProvMonitor was able to capture implicit provenance and evaluated how the gathered provenance can aid the scientists' workflow analysis. Some different aspects were considered regarding the research and secondary questions of this work: (i) is ProvMonitor able to capture implicit provenance? Is it possible to make existing SWfMS aware of implicit provenance? (ii) is ProvMonitor able to version intermediate files? Does gathering of intermediate file content improve implicit provenance analysis? (iii) does the implicit provenance gathering improve the perception about the impact of implicit data flow over activities executions and experiment results? and (iv) does the implicit provenance gathering contribute to a more precise analysis about the experiment execution? For this evaluation, we used adaptations of queries already used to evaluate SciPhy (OCAÑA *et al.*, 2014). These queries were adaptations of the First and Second Provenance Challenges. We have just considered queries related to data file production, on situations that are useful to SciPhy, since this may involve implicit data flows and is where approaches that do not gather implicit provenance usually fail. In the efficiency evaluation, we analyze the performance and the overhead introduced by ProvMonitor to SciCumulus

regarding time-consumption and storage overheads during SciPhy execution over SciCumulus. The objective is to answer the secondary question: does the overhead imposed by the implicit provenance gathering preclude its adoption?

All experiments presented in this section were performed in the Amazon EC2 environment using Amazon's large VMs (EC2 ID: m1.large – 7.5 GB RAM, 850 GB storage, 2 cores). Each instantiated VM in the phylogenetic experiments presented in this work uses Linux Cent OS 5 (64-bit). To execute SciPhy, our experiments use a dataset of multi-fasta files of protein sequences extracted from RefSeq release 48. This dataset is formed by 200 multi-fasta files and each multi-fasta file is constituted by an average of 10 biological sequences. To perform phylogenetic analysis, each input multi-fasta file is processed using MAFFT version 6.857, ModelGenerator version 0.85, and RAxML-7.2.8-ALPHA. For this case study, ProvMonitor operated with the PostgreSQL database, since this is the database used by SciCumulus.

In this chapter, we first introduce SciCumulus and then present the specification of the SciPhy workflow. Then, we detail how SciPhy was modeled and executed in SciCumulus. Next, we present the effectiveness and the efficiency analysis of the performance of ProvMonitor while capturing implicit provenance.

3.2 SCICUMULUS WORKFLOW ENGINE

SciCumulus is an important component since it executes the workflows to be monitored by ProvMonitor. It is important to highlight that other workflow execution engines, besides SciCumulus, can also benefit from the proposed approach.

SciCumulus is an engine that manages the parallel execution of scientific workflows in clouds, such as Amazon EC2[3] or Microsoft Azure[4]. Four tiers compose SciCumulus architecture: Client tier, Distribution tier, Execution tier, and Data tier. Figure 12 summarizes a high-level conceptual architecture.

SciCumulus client tier is responsible for initiating the execution of workflow activities in the cloud. The components of this tier are deployed in the scientist desktop or within a SWfMS, such as VisTrails (CALLAHAN *et al.*, 2006) or Kepler (ALTINTAS; BARNEY; JAEGER-FRANK, 2006). The distribution tier manages the execution of the several cloud activities. This tier is responsible for creating the scheduling plan for the cloud activities to be executed in the different virtual machines that are part of the execution tier. The execution tier

[3] http://aws.amazon.com/pt/ec2/ - Jun/2014
[4] https://www.windowsazure.com/en-us/ - Jun/2014

is responsible for invoking executable codes (*i.e.*, programs that are part of the workflow) in the several allocated virtual machines. Finally, the data tier has the provenance repository and the file system that stores all files produced and consumed during a trial.

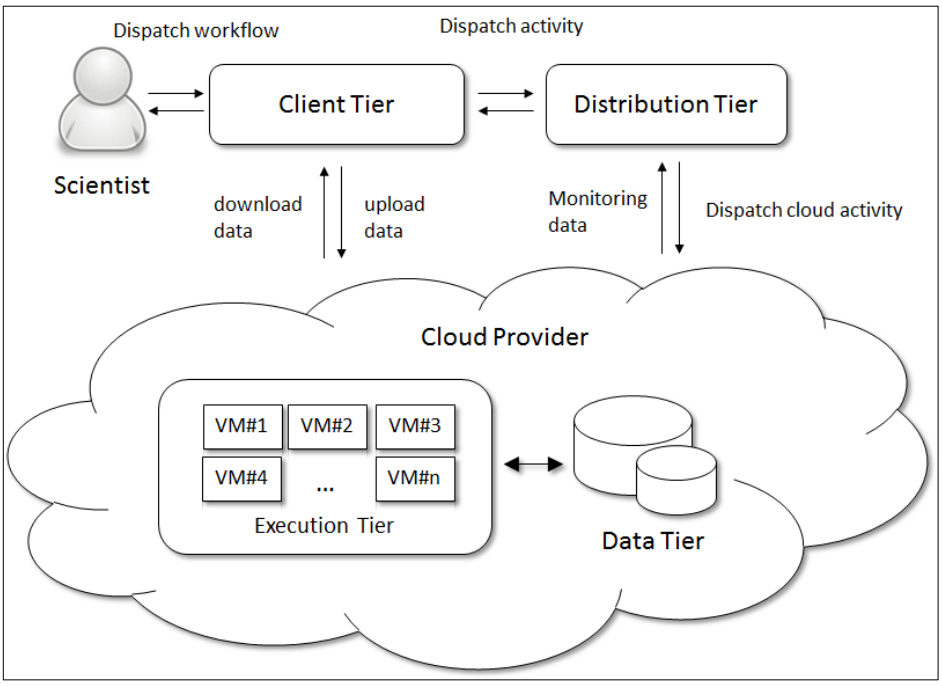


Figure 12: SciCumulus Conceptual Architecture (GONÇALVES *et al.*, 2012)

To understand SciCumulus architecture it is important to understand an important concept that is cloud activity. In SciCumulus, algebraic operators that consume and produce sets of tuples (relations) rules workflow activities, as proposed by (OGASAWARA *et al.*, 2011). This way, SciCumulus uses a declarative workflow representation in which each workflow activity produces and consumes relations. Since in parallel trials the same activity can execute several times varying parameters and input data, SciCumulus has to manage these several executions. To support such execution, SciCumulus is based on the concept of cloud activity, inspired by database tuple activations (BOUGANIM; FLORESCU; VALDURIEZ, 1996). The concept of cloud activity is critical to make a workflow representation able to execute in large-scale cloud environments. A cloud activity is a self-contained object that stores all information needed for a specific executable (*i.e.*, the program to be invoked and the portion of data to be consumed) to execute an activity at any virtual machine. Cloud activities contain the finer unit of data needed by an activity to execute (BOUGANIM; FLORESCU; VALDURIEZ, 1996). Cloud activities are processed in SciCumulus following a 3-step procedure: instrumentation – where the parameter values are gathered to prepare the program invocation; program invocation – where the executable code is effectively executed; and output extraction – where result data is extracted and stored in the provenance database.

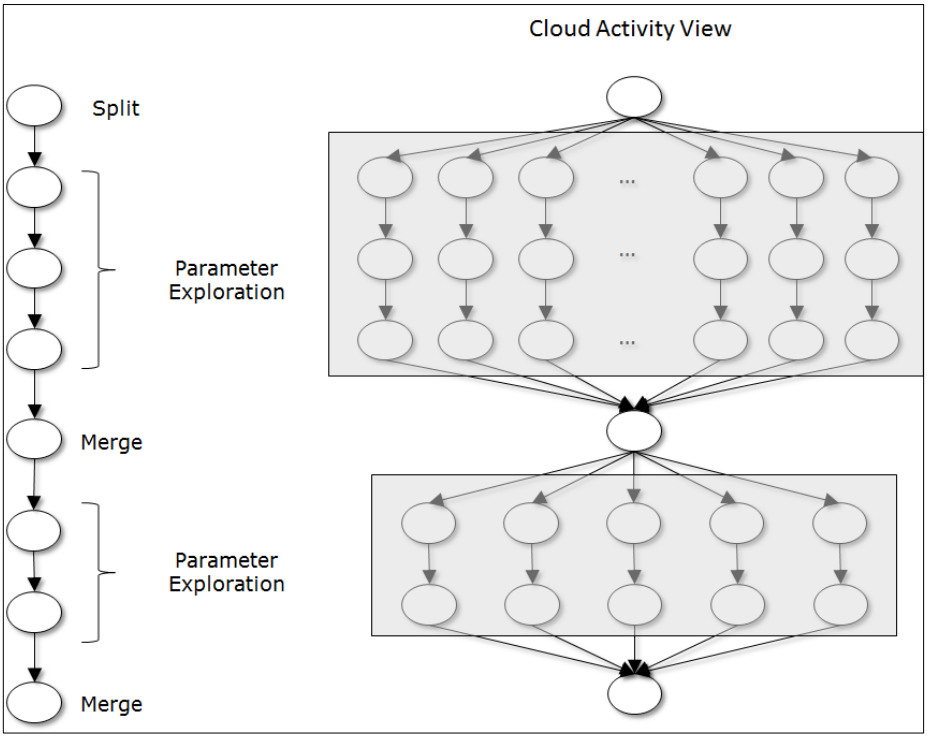


Figure 13: Example of Cloud Activity generation (OLIVEIRA, 2012)

Figure 13 presents a generic example of the cloud activity generation. In this example, the workflow starts with a split activity and then there are two merge activities. These three activities cannot execute in parallel. However, the other activities can execute in parallel, thus SciCumulus creates a different cloud activity for each different input data file to be consumed (Cloud Activity View). The generated cloud activity is then scheduled to execute in the several allocated virtual machines. For more information about SciCumulus please refer to (OLIVEIRA *et al.*, 2010).

3.3 SCIPHY

SciPhy is a scientific workflow for phylogenetic analysis. Phylogenetic analysis aims at producing phylogenetic trees to represent existing evolutionary relationships. The analysis is done by consuming several input data files containing a set of DNA, RNA, or amino acid sequences. It requires a complex workflow composed by several data and computing-intensive activities that may consume considerable time to produce the desired result. SciPhy is illustrated in Figure 14, where each rectangle indicates an activity, solid lines represent data dependency between activities, and dashed lines represent consumed and produced data products.

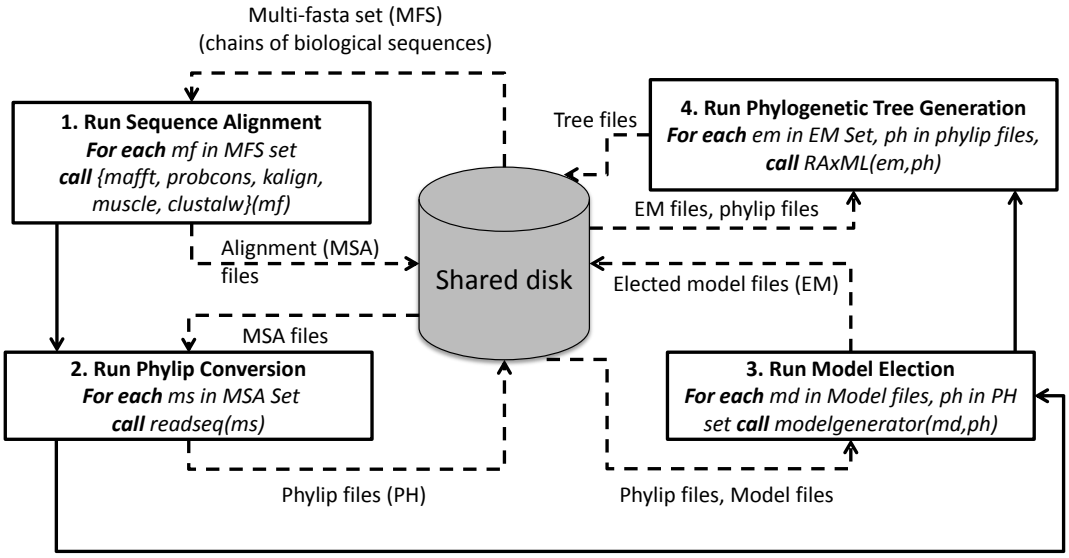


Figure 14. SciPhy workflow adapted from Gonçalves et al. (2012)

The first activity of the workflow (Figure 14) consumes a set of multi-fasta files (MFS) and produces a set of alignment files (MSA). Similarly, the second activity consumes a set of MSA and produces phylip files (PH). However, there are more steps involved in the workflow trial than the ones represented in the workflow specification. For example, Activity 1 produces MSA files but also modifies some of the MFS input files, which are consumed by Activity 2. This data dependency (MFS as input to Activity 2) is not explicitly represented in the workflow specification, nor is the changes to MFS. In fact, there are different versions of MFS files involved in this experiment. Every time Activity 1 modifies one MFS file, it generates a new version of it. The problem is that old versions may be overwritten by new versions, and provenance data related to this change is lost. ProvMonitor solves this problem by storing all the versions of a data item, thus keeping the history of changes. Since scientists do not specify the modifications performed by Activity 1 in the MFS files and their posterior uses in Activity 2, this scenario characterizes the aforementioned implicit data flow.

SciPhy workflow is composed of four main activities: Multiple Sequence Alignment (MSA) construction, MSA format conversion, a search for the best evolutionary model, and phylogenetic tree construction. They respectively execute the following bioinformatics applications: MAFFT (KATOH; TOH, 2010), ReadSeq (GILBERT, 2003), ModelGenerator (KEANE *et al.*, 2006), and RAxML (ROKAS, 2011). Figure 15 presents the SciPhy conceptual view. The first activity of SciPhy constructs an individual MSA with the MAFFT MSA program. MAFFT receives a multi-fasta file as an input from a set of given multi-fasta files (*e.g.*, from the RefSeq (PRUITT *et al.*, 2009) biological database) and then produces an MSA file as output. In the second activity, the MSA is converted to the PHYLIP format

(FELSENSTEIN, 1989, 2005) using ReadSeq and then tested in the third activity to find the best evolutionary model using ModelGenerator. Both the individual MSA and the evolutionary model are necessary for the fourth activity to generate phylogenetic trees using RAxML.

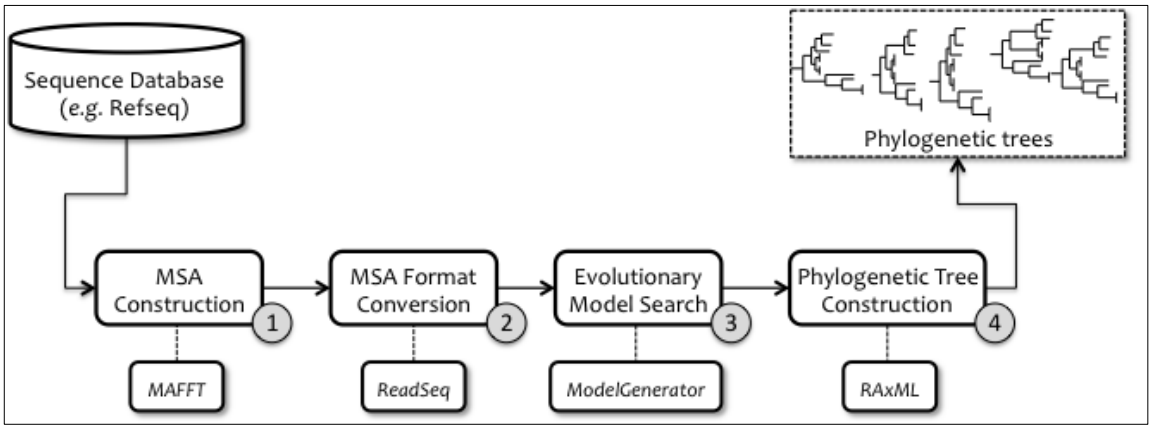


Figure 15: SciPhy workflow conceptual view adapted from Ocaña et al. (2011)

Thus, SciPhy allows scientists to:

i. Infer phylogeny and to delineate evolutionary relationships between the species (taxa) used in the study and;

ii. Explore genomes for searching/inferring new candidates for drug targets, such as enzymes, based on the tree taxa position.

The large amount of reliable trees obtained during exploratory analysis can represent valuable phylogenetic information between species and genomes. The SciPhy workflow is much more complex than the conceptual view detailed in Figure 14 and Figure 15. It can be executed multiple times (one trial for each set of parameter values) as required, depending on the parameter sweep strategy defined in the workflow. Thus, all trials produce a huge amount of versions of different formats of files. Ocaña et al. (2011) present more details about the SciPhy workflow.

3.4 EFFECTIVENESS EVALUATION

We processed 150 of the aforementioned 200 multi-fasta files using SciCumulus. Based on the results, we performed two analyses. The first one verifies if ProvMonitor was able to answer the proposed queries. The second shows how implicit provenance management can aid scientists to better explore their results for assessing biological hypotheses. The first and second Provenance Challenge (PC1 and PC2) (MILES, 2006a, 2006b) defined a series of provenance queries for Pan-STARRS workflow (ARAGON; RUNGE, 2009). We describe

three adapted provenance queries (in English and SQL), which are executable over our database schema, to demonstrate querying capabilities of ProvMonitor. We also discuss how each one of the answers can aid scientists.

Query 1 (Q1). During the experiment, SciPhy workflow is executed more than once. Several intermediate files are commonly generated by SciPhy workflow activities. If the scientist runs a trial by varying the bootstrap replication parameter (*e.g.*, 50, 100), it could be interesting to compare the trees obtained with this variation to indicate if different bootstraps can influence in the trees' topology or bootstrap values and in what manner. Thus, varying bootstrap replication values in RAxML parameters can lead to differences in phylogenetic trees, which consequently could influence biological or computational inferences. Does ProvMonitor find the differences between data products generated by several workflow trials, for each activity? This is an adaptation of Query 7 of the PC1, which aims at comparing two or more trials (inter-trial analysis). ProvMonitor is able to answer this question, and it uses the SQL query shown in Table 2 for this, together with diff operations in Git.

Table 2: Q1 SQL and expected results

SQL: SELECT PATH, ID_COMMIT, COMMIT_TIME FROM ELEMENT_EXECUTION_COMMIT;
ANSWER: This query returns the commit hashes of every commit done by every activity of every trial, allowing us to locate all file versions created after each one of the activities. A diff operation in Git over the commits with these ids answers the second part of the query.

This answer allows scientists to compare results among different trials, comparing the trees obtained with the bootstrap replication variation and identifying when the variation leaded to differences in phylogenetic trees. For example, Figure 16 shows phylogenetic trees constructed with different bootstrap replication values. Tree A was built with bootstrap of 50 and Tree B with 100. Both trees show the same topology, but they present different bootstrap replication values, as Tree B presents lower values than Tree A. For example, they present the most representative monophyletic clade formed by species 4, 7, 6, and 2. Starting from the most external to the terminal nodes, Tree A presents bootstraps of 100, 84, and 96, while Tree B presents bootstraps of 99, 80, and 94. Thus, varying bootstrap replication values in RAxML parameters can lead to differences in phylogenetic trees, which consequently could influence biological or computational inferences.

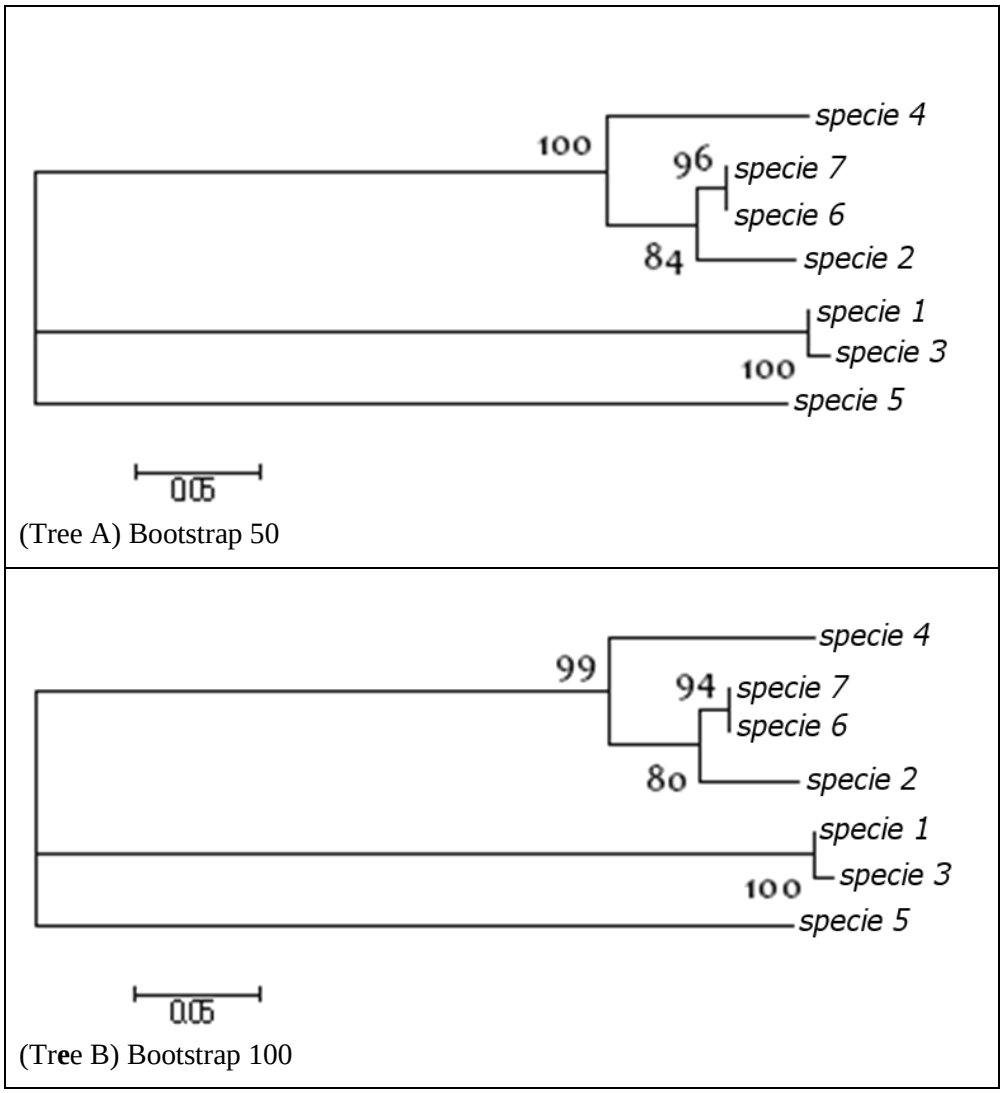


Figure 16: Phylogenetic trees with bootstrap values related to the "Phylogenetic Tree Construction" activity adapted from Ocaña et al. (2013)

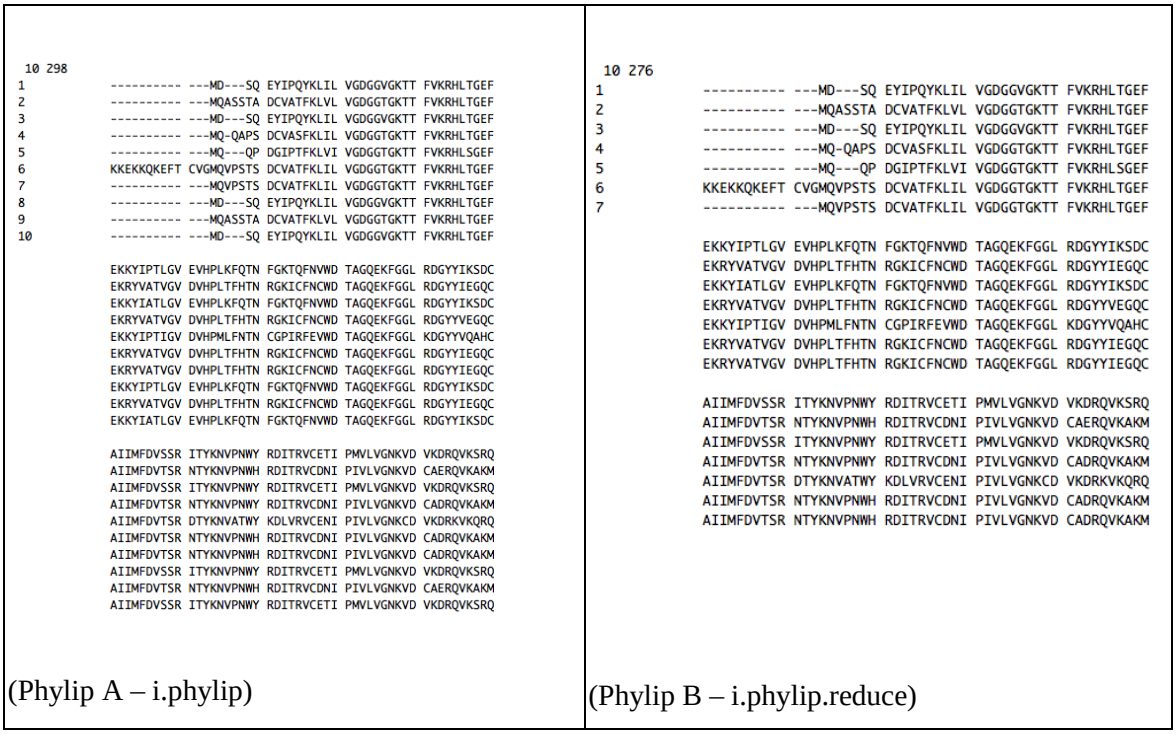
Query 2 (Q2). In biological scenarios, knowledge about all generated files is an important issue, since it allows scientists to have the control of their trial process. For example, RAxML may automatically reduce phylip files. This decision usually depends on some features characterizing the multi-fasta files, *e.g.*, due the presence of almost equal fasta sequences or if any fasta sequence is too different to the rest. In this scenario, RAxML automatically produces the "i.phylip.reduce" file from the original "i.phylip". Since the modification over the "i.phylip" file was not defined by any parameter in SciPhy, scientists cannot predict this behavior, as it is hard-coded in RAxML program. As RAxML reduces the original "i.phylip" file, a different phylogenetic tree can be obtained by using the "i.phylip.reduce" file. For a given constructed phylogenetic tree, what was the input phylip file from which the tree was produced? Can ProvMonitor detect this behavior and point out

the difference between the “i.phylip” and “i.phylip.reduce” files? This query is similar to Query 5 of PC1 and PC2. The query shown in Table 3 answers this question.

Table 3: Q2 SQL and expected results

SQL: SELECT EFS.FILE_PATH, EFS.FILE_ACCESS_TYPE, EFS.FILE_ACCESS_TIME, EC.ID_COMMIT FROM ELEMENT_EXECUTION_FILE_ACCESS EFS LEFT JOIN ELEMENT_EXECUTION_COMMIT EC ON EFS.ELEMENT_PATH LIKE EC.PATH WHERE UPPER(EFS.FILE_PATH) LIKE UPPER('%i.phylip.reduced%')
ANSWER: This query returns all “i.phylip.reduced” creations and changes, the timestamp when it occurred and the *commit* hashes to the file version, allowing us to recover the related files and compare them.

This answer opens some interesting opportunities allowing scientist to compare and identify changes on files at the same trial (intra-trial analysis). Figure 17 shows an example of an “i.phylip” file and a correspondent “i.phylip.reduce” file. It is possible to identify the exclusion of some lines from the original file (Phylip A), leading to a file with a reduced number of lines (Phylip B). This perception, gives the scientist opportunity to identify exactly which lines were removed, helping to understand the effect of a given activity and allowing exploratory tasks, for example, evaluating the effects of computing the removed lines.



```
 10 298
1          ---------- ---MD---SQ EYIPQYKLIL VGDGGVGKTT FVKRHLTGEF
2          ---------- ---MQASSTA DCVATFKLVL VGDGGTGKTT FVKRHLTGEF
3          ---------- ---MD---SQ EYIPQYKLIL VGDGGVGKTT FVKRHLTGEF
4          ---------- ---MQ-QAPS DCVASFKLIL VGDGGTGKTT FVKRHLTGEF
5          ---------- ---MQ---QP DGIPTFKLVI VGDGGTGKTT FVKRHLSGEF
6          KKEKKQKEFT CVGMQVPSTS DCVATFKLIL VGDGGTGKTT FVKRHLTGEF
7          ---------- ---MQVPSTS DCVATFKLIL VGDGGTGKTT FVKRHLTGEF
8          ---------- ---MD---SQ EYIPQYKLIL VGDGGVGKTT FVKRHLTGEF
9          ---------- ---MQASSTA DCVATFKLVL VGDGGTGKTT FVKRHLTGEF
10         ---------- ---MD---SQ EYIPQYKLIL VGDGGVGKTT FVKRHLTGEF

           EKKYIPTLGV EVHPLKFQTN FGKTQFNVWD TAGQEKFGGL RDGYYIKSDC
           EKRYVATVGV DVHPLTFHTN RGKICFNCWD TAGQEKFGGL RDGYYIEGQC
           EKKYIATLGV EVHPLKFQTN FGKTQFNVWD TAGQEKFGGL RDGYYIKSDC
           EKRYVATVGV DVHPLTFHTN RGKICFNCWD TAGQEKFGGL RDGYYVEGQC
           EKKYIPTIGV DVHPMLFNTN CGPIRFEVWD TAGQEKFGGL KDGYYVQAHC
           EKRYVATVGV DVHPLTFHTN RGKICFNCWD TAGQEKFGGL RDGYYIEGQC
           EKRYVATVGV DVHPLTFHTN RGKICFNCWD TAGQEKFGGL RDGYYIEGQC
           EKKYIPTLGV EVHPLKFQTN FGKTQFNVWD TAGQEKFGGL RDGYYIKSDC
           EKRYVATVGV DVHPLTFHTN RGKICFNCWD TAGQEKFGGL RDGYYIEGQC
           EKKYIATLGV EVHPLKFQTN FGKTQFNVWD TAGQEKFGGL RDGYYIKSDC

           AIIMFDVSSR ITYKNVPNWY RDITRVCETI PMVLVGNKVD VKDRQVKSRQ
           AIIMFDVTSR NTYKNVPNWH RDITRVCDNI PIVLVGNKVD CAERQVKAKM
           AIIMFDVSSR ITYKNVPNWY RDITRVCETI PMVLVGNKVD VKDRQVKSRQ
           AIIMFDVTSR NTYKNVPNWY RDITRVCDNI PIVLVGNKVD CADRQVKAKM
           AIIMFDVTSR DTYKNVATWY KDLVRVCENI PIVLVGNKCD VKDRKVKQRQ
           AIIMFDVTSR NTYKNVPNWH RDITRVCDNI PIVLVGNKVD CADRQVKAKM
           AIIMFDVTSR NTYKNVPNWH RDITRVCDNI PIVLVGNKVD CADRQVKAKM
           AIIMFDVSSR ITYKNVPNWY RDITRVCETI PMVLVGNKVD VKDRQVKSRQ
           AIIMFDVTSR NTYKNVPNWH RDITRVCDNI PIVLVGNKVD CAERQVKAKM
           AIIMFDVSSR ITYKNVPNWY RDITRVCETI PMVLVGNKVD VKDRQVKSRQ
```

(Phylip A – i.phylip)

```
 10 276
1          ---------- ---MD---SQ EYIPQYKLIL VGDGGVGKTT FVKRHLTGEF
2          ---------- ---MQASSTA DCVATFKLVL VGDGGTGKTT FVKRHLTGEF
3          ---------- ---MD---SQ EYIPQYKLIL VGDGGVGKTT FVKRHLTGEF
4          ---------- ---MQ-QAPS DCVASFKLIL VGDGGTGKTT FVKRHLTGEF
5          ---------- ---MQ---QP DGIPTFKLVI VGDGGTGKTT FVKRHLSGEF
6          KKEKKQKEFT CVGMQVPSTS DCVATFKLIL VGDGGTGKTT FVKRHLTGEF
7          ---------- ---MQVPSTS DCVATFKLIL VGDGGTGKTT FVKRHLTGEF

           EKKYIPTLGV EVHPLKFQTN FGKTQFNVWD TAGQEKFGGL RDGYYIKSDC
           EKRYVATVGV DVHPLTFHTN RGKICFNCWD TAGQEKFGGL RDGYYIEGQC
           EKKYIATLGV EVHPLKFQTN FGKTQFNVWD TAGQEKFGGL RDGYYIKSDC
           EKRYVATVGV DVHPLTFHTN RGKICFNCWD TAGQEKFGGL RDGYYVEGQC
           EKKYIPTIGV DVHPMLFNTN CGPIRFEVWD TAGQEKFGGL KDGYYVQAHC
           EKRYVATVGV DVHPLTFHTN RGKICFNCWD TAGQEKFGGL RDGYYIEGQC
           EKRYVATVGV DVHPLTFHTN RGKICFNCWD TAGQEKFGGL RDGYYIEGQC

           AIIMFDVSSR ITYKNVPNWY RDITRVCETI PMVLVGNKVD VKDRQVKSRQ
           AIIMFDVTSR NTYKNVPNWH RDITRVCDNI PIVLVGNKVD CAERQVKAKM
           AIIMFDVSSR ITYKNVPNWY RDITRVCETI PMVLVGNKVD VKDRQVKSRQ
           AIIMFDVTSR NTYKNVPNWY RDITRVCDNI PIVLVGNKVD CADRQVKAKM
           AIIMFDVTSR DTYKNVATWY KDLVRVCENI PIVLVGNKCD VKDRKVKQRQ
           AIIMFDVTSR NTYKNVPNWH RDITRVCDNI PIVLVGNKVD CADRQVKAKM
           AIIMFDVTSR NTYKNVPNWH RDITRVCDNI PIVLVGNKVD CADRQVKAKM
```

(Phylip B – i.phylip.reduce)

Figure 17: Phylip files related to the “MSA Construction” activity

Indeed, this answer also allows identifying which one of the files was consumed by the next activity: the original file or the reduced one. This is important, since that, besides the generation of a different file (Phylip B), it may not exist any guarantee of which of the files would be consumed on the remaining of the trial. Thus, identifying the consumed file helps to understand the effect and influence of intermediate files upon the following trial execution.

Query 3 (Q3). Programs MAFFT and ReadSeq consume the same fasta file. However, the "MSA Construction" activity can behave in two different ways: (i) it only consumes the input fasta files or (ii) it modifies the input fasta files by removing sequences to provide load balancing. As some sequences were removed from the fasta file, the MAFFT program in the "MSA Construction" activity constructs a reduced MSA instead of the total MSA with all original fasta sequences. The obtained result directly influences in the following "MSA Format Conversion" activity, because it will use the reduced MSA instead of the total MSA. Then, depending on the chosen behavior, the "MSA Format Conversion" activity may consume the original or the overwritten fasta file. Is ProvMonitor able to detect all modifications? Where are they performed? This is the only query that has no equivalent in PC1 and PC2, since this query is related to implicit provenance. Query 3 presents the traditional implicit provenance case. The query shown in Table 4 answers this question.

Table 4: Q3 SQL and expected results

SQL: SELECT * FROM ELEMENT_EXECUTION_COMMIT EC WHERE EC.PATH LIKE '%TRIAL_ID%ACTIVITY_ID%';
ANSWER: This query returns the *commit* hashes of every *commit* after each specified activity (ACTVITY_ID) on each specified trial (TRIAL_ID), thus allowing to compare the files before and after the activity that changed it with diff algorithms (textual or specialized diffs).

With this answer, the scientist may identify exactly the version of a file used to generate other files. Figure 18 shows an example of such perception. The "MSA Construction" activity changed the original "fasta" file (Fasta A) by reducing sequences. Then, as some sequences were removed (Figure 18 Fasta B) from the "fasta" file, the MAFFT program in the "MSA Construction" activity constructs a reduced MSA (Figure 18 Alignment B) instead of the total MSA with all original fasta sequences (Alignment A).

Figure 18: Fasta and alignment files content to the "MSA Construction" activity

It is interesting to highlight that ProvMonitor capability of identify every changed file after an activity execution allows scientist to evaluate the workflow specification. It is possible to identify if every file specified at the workflow specification is actually accessed, and if every file accessed or created are specified at the workflow specification. Thus, by contrasting files access (read or write) described as prospective provenance with files accesses (read or write) gathered as retrospective provenance via SQL, it is possible to identify every workflow activity that participates in an implicit data flow, *i.e.*, that produce implicit provenance, just querying the provenance database.

3.5 EFFICIENCY EVALUATION

We conducted two different experiments to analyze the influence of ProvMonitor in the workflow trial: the first one identifies the execution time overhead and the second one identifies the overhead imposed in storage. Since ProvMonitor uses instrumented activities in SciCumulus (in this experiment, instrumentation was performed manually), it introduces a certain overhead in each cloud activity execution. To analyze the impact of ProvMonitor we executed the SciPhy workflow consuming 150 of the 200 input multi-fasta files set in a single virtual machine in Amazon EC2 cloud, using the third isolation strategy (workspace per activity with a branch per activity). We measured the needed time to execute SciPhy in SciCumulus with and without using ProvMonitor. As expected, ProvMonitor instrumentations in SciCumulus cloud activities impact in the performance, turning them more time-consuming. However, the impact of ProvMonitor depends on the type of activity in the workflow. In SciPhy, we have two types of activities: short-term and long-term activities. The first two activities are short-term activities. Each execution of these activities demands few seconds to finish. On the other hand, the last two activities are long-term activities. Each cloud activity associated to these activities demands at least 10 minutes to finish.

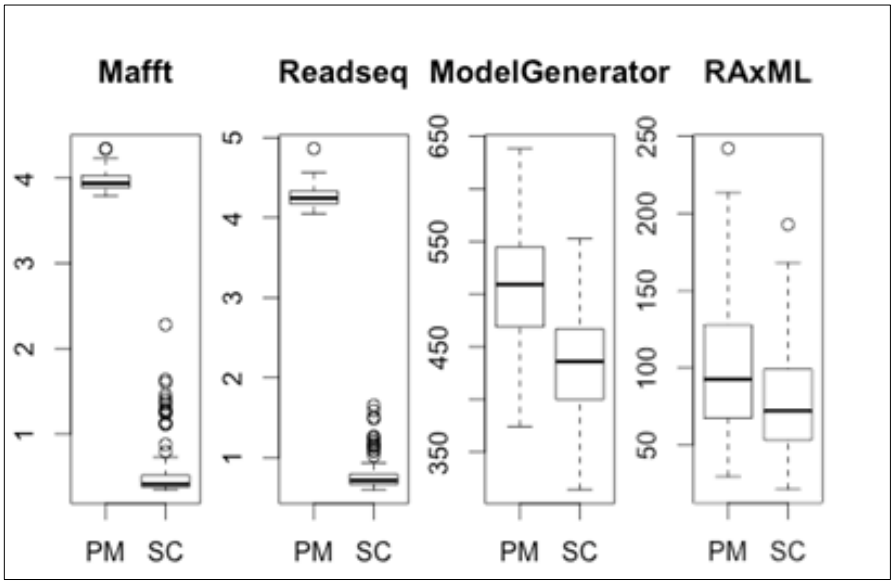


Figure 19: Boxplot of ProvMonitor overhead over SciPhy activities

Figure 19 shows boxplots of this experiment, where the *x-axis* represents the cloud activity execution with (PM) or without (SC) ProvMonitor and the *y*-axis represents its execution time distribution in seconds for a given mean (μ) and standard deviation (σ). This is also shown on Table 5 - using ProvMonitor (labeled ProvMonitor) and without using ProvMonitor (labeled SciCumulus). In average, each short-term cloud activity using

ProvMonitor needs from 5.39 to 7.08 more time than when we are not using it. Although it looks like a high overhead, actually, ProvMonitor adds nearly constants 4 seconds (to prepare the workspace – I/O on files, Git manipulation, and gathering and storing metadata) in each short-term activity. Since each short-term cloud activity executes for less than one second (0.74 seconds in average), the impact is large for this kind of activity.

When we analyze the impact in the long-term cloud activities, we can state that, in average, each long-term activity using ProvMonitor needs from 1.16 to 1.28 more time than when we are not using ProvMonitor. Here the impact of ProvMonitor in the cloud activities is not critical when compared to the necessary time to execute the program by the cloud activity.

Table 5: Mean and standard deviation of Cloud activities execution times for 100 input files

Workflow activities	ProvMonitor		SciCumulus	
	μ	σ	μ	σ
MAFFT	3.95	0.10	0.56	0.36
ReadSeq	4.26	0.12	0.79	0.21
MG	506.29	56.19	433.45	51.18
RAxML	101.76	46.88	79.68	37.64

Table 5 evidences some interesting behaviors related to the execution time overhead. For example, the standard deviation of activities MAFF and ReadSeq are smaller with ProvMonitor. This may be explained by the strategy used to compose the workflow. In this experiment, to simplify the workspace management, the workspace contains all the files used by all trials. Then, ProvMonitor introduces a nearly constant 4 seconds of time overhead per activation in this activities. Without ProvMonitor SciCumulus still needs to copy files however, the files are copied by demand instead of the entire workspace. As the number and size of the files may vary, this explains a higher standard deviation without ProvMonitor.

On the other hand, MG and RAxML activities present a higher standard deviation with ProvMonitor. At a first look, this may not make sense. However, these two activities are the last ones to execute, thus we have to consider that during a trial, environment degradation may occur (*e.g.*, less disk space available) caused by the previously executed activities. For example, when the first MG executes for 100 input files, 100 MAFFT and 100 ReadSeq activities had already been executed. Therefore, in practice, the first MG is the 201st activity to be executed. Although this analysis appears to make sense, there are many other factors beyond our control when running on Virtual Machines (VM) over management of specific

services such as Amazon, that do not guarantee exactly the same performance (but a performance inside a range). For example, a VM may be physically migrated during execution (live migration), which leads to environment degradation. Such factors make it difficult to find the precise reasons for the aforementioned standard deviation differences with and without ProvMonitor.

Other interesting behavior is the execution times variation of MG and RAxML activities with ProvMonitor being larger than the aforementioned "nearly constants" 4 seconds. Indeed, being the last activities to be executed, they execute over a workspace composed by the new intermediate files and metadata generated by the previous activities during the workflow trial. The new workspace composition being compound by a larger amount of files and metadata may consume more time to be cloned, explaining a higher execution time variation. Again, the environment degradation may be the main influence of such divergence.

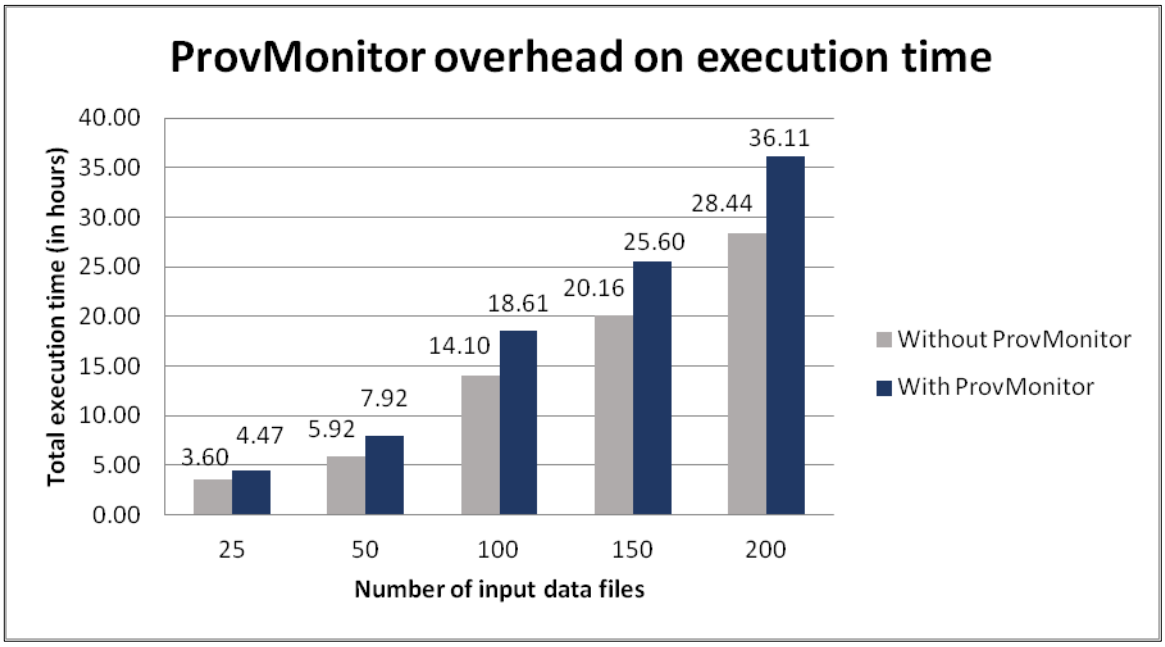


Figure 20: ProvMonitor overhead in the entire workflow trial time

Even though the added time in each cloud activity may be small in absolute values, we have to analyze the impact in the overall trial. We have varied the amount of input data files to be processed in each trial. We executed SciPhy consuming 25, 50, 100, 150, and 200 multi-fasta files using ProvMonitor and without using ProvMonitor. Figure 20 presents the total execution time of SciPhy in the *y-axis* and the amount of processed data files in *x-axis*. By analyzing Figure 20, we can state that in this scenario ProvMonitor severely affected the

workflow trial. For example, when we processed 200 multi-fasta files using ProvMonitor we needed 7 more hours than when we did not use ProvMonitor (26% increase).

Actually, this overhead tends to increase according to the size of the workflow, *i.e.*, the more cloud activities to process, the more time ProvMonitor needs. In the scenario presented in Figure 20, the total execution time difference is harvest by the short-term activities. Since each short-term activity is 7 times slower in average, the execution of the first two activities of the workflow produced a negative impact in the overall execution. If we only consider the performance aspect, the use of ProvMonitor is more suitable for workflows that present medium and long-term activities. However, when we consider the benefits of analyzing implicit and hidden provenance, this performance loss is justified.

We also analyzed the storage overhead of ProvMonitor. We measured the necessary disk space for executing SciCumulus with ProvMonitor. In addition, we also measured the necessary disk space for the resultant central repository (that contains all versions of all files involved in the trial). To analyze this overhead we executed SciPhy consuming 25, 50, 100, and 150 multi-fasta files – these set of files constitute a small dataset (less than 1 MB – that goes to less than 5 MB under Git management). Table 6 presents the disk space needed during and after execution for each number of processed input data files.

Results show that, even for the small dataset, ProvMonitor needed more than 4 GB during the trial for processing 150 input data files. The cause is ProvMonitor's execution that replicates the input data files for each cloud activity. This is done due to the chosen isolation strategy, the workflow characteristics, and the SciCumulus behavior, since it stores data of all trials concurrently - "*first activity first*" execution - that consumes more disk space. In addition, Git replicates several data files to accelerate the version control. However, SciCumulus behavior is probably one of the main reasons of this overhead, since that removing intermediate workspaces after use, or at least, creating less concurrent workspaces at the same time would lead to a significant decrease of the amount of disk space needed.

Although the consumed disk space is acceptable if we consider the today's disk capacity and the final size of the repository, ProvMonitor introduced storage overhead when executing the workflow. This had a reduced impact since we used the Amazon S3 storage where each bucket has 273 TB of data, which is considerably large even for large-scale workflow executions. For scientific workflows that consume gigabytes of input datasets the necessary disk space could be an issue, unless large-scale storages such as S3 are used, or a smaller number of trials is executed simultaneously.

Table 6: Disk space overhead with ProvMonitor

Number of input multi-fasta files	SciCumulus without ProvMonitor		SciCumulus with ProvMonitor		
	Disk space during execution (MB)	Resultant output (MB)	Disk space during execution (MB)	Resultant central repository (MB)	Resultant central repository after Git optimization (MB)
25	5.6	4.9	770	4.1	2.0
50	12.1	9.8	1,600	5.5	2.5
100	27.4	19.5	3,100	8.6	3.4
150	63	27.3	4,700	11	4.5

Although Git provides an optimization mechanism ("*gc --aggressive*" command) that reduces the amount of necessary disk space (last column of Table 6), this optimization can only be performed after the workflow completion. This way, it is interesting to test other strategies for larger experiments. One option is to use different workspace and branch strategies, for example, exploring other VCS capabilities such as bare repositories or avoiding data replication on each trial. Another option is to explore environments that could benefit from VCS clone optimizations through hard links instead of replicating data. Another one is to explore Git optimizations through shared repositories.

Table 7 presents some results of using such Git optimizations on a Linux environment with support to hard links. For the evaluation, we used a repository with 100 input files and 100 branches. The repository was measured before a "*git gc --aggressive*" (column Input) consuming 2.2 MB of disk space and after a "*git gc --aggressive*" (column Input GC) consuming 1.6 MB of disk space. The "*gc*" command produces an optimization on Git repository storage, by compressing the history files and by maximizing objects reuse (Git identifies similar files storing the content once and creating different references for the same data objet into the repository). The line labeled as "Normal clone" shows the results of cloning repositories without optimizations, such as what was done during the evaluation of ProvMonitor presented in this chapter. The line labeled as "Hard links" shows the results of cloning repositories by benefiting of hard links for the original repository instead of copying the files. With hard links, files are copied only when modified, since in this moment the

modified file is different from the referenced one. Finally, the line labeled as “Share” shows the results of cloning without copying the Git repository. The target repository objects are references to the source repository. Only the files of the workspace are copied.

Results on Table 7 show that both strategies (hard links or shared repository) present optimizations on storage consumption. However, the shared repository strategy also seems to optimize the execution time consumed during the clone operation. Such results indicate that, when running workflows on local files, or even on storages, without cloning over networks, these optimizations mechanisms of Git may help minimize the overhead imposed by ProvMonitor during the workflow trial.

Table 7: Git clone optimizations on execution time and storage

Cloning optimizations	Input (2.2 MB)		Input GC (1.6 MB)	
	Time	Storage	Time	Storage
Normal clone	0.181s	2.2 MB	0.061s	1.6 MB
Hard links	0.063s	1.6 MB	0.068s	1.3 MB
Share	0.053s	1.6 MB	0.045s	1.3MB

Results on Table 7 also show that ProvMonitor’s overhead is not imposed exclusively by the VCS usage. Indeed, there are other steps performed by ProvMonitor that contribute to the overhead, such as accessing file metadata to identify files accesses and communication with database. Although performance optimizations were not one of the prototype development premises, results showed that some optimizations should be addressed to minimize the overhead. For example, a more efficient strategy to identify files access (read access) instead of sequentially reading metadata and independent threads to database communication are some of such possible optimizations.

Finally, it is important to highlight that besides consuming more disk space during the workflow trial, ProvMonitor optimizes the storage of the results (columns 3, 5, and 6 of Table 6). This is because SciCumulus alone does not control files from a trial. Therefore, the scientist needs to specify, in the workflow, the copy of files to an output directory. This opens two possible problems: overwrite (if the same output directory is used for every trial) or files replication (if different directories are used for different trials). ProvMonitor optimizes that by versioning files and identifying the trial that generated each files version.

3.6 FINAL REMARKS

This chapter presented the evaluation of the proposed approach. Results showed the effectiveness of the approach on the implicit provenance gathering, and some benefits of

implicit provenance gathering on the experiment analysis. However, results also highlighted that some overhead may be imposed during the implicit provenance gathering, thus, the ideas proposed by this work has to be carefully applied to avoid that the overhead overcomes the benefits of implicit provenance gathering.

The next chapter presents an overview on the state of the art on implicit provenance gathering and VCS usage on provenance gathering.

CHAPTER 4 – RELATED WORK

4.1 INTRODUCTION

Implicit provenance has received little attention in the literature. Actually, implicit provenance of implicit data flows is not considered by existing provenance management approaches (CALLAHAN *et al.*, 2006; MARINHO *et al.*, 2011b; MOUALLEM *et al.*, 2009; OLIVEIRA *et al.*, 2010). OS-based provenance gathering approaches (FREW; METZGER; SLAUGHTER, 2008; MUNISWAMY-REDDY *et al.*, 2006) are the ones that capture part of this provenance. However, generally speaking, they do not relate prospective and retrospective provenance and may produce a large amount of provenance not directly related to the workflow trial on a fine-grained and low-level representation such as OS process IDs. This large amount of information may be distractive or even prohibitive to scientists' analysis.

Following, we present some approaches that relate to ProvMonitor in some level. They were identified through a literature review study. This literature review aims at answering a research question (RQ) and some secondary questions (SQ) described below.

RQ: How to capture implicit provenance? How to deal with it?

SQ1: How to gather implicit provenance using configuration management?

SQ2: How to use version control systems to support provenance gathering?

The research started from a set of already known papers (CHIRIGATI; SHASHA; FREIRE, 2013; DAVISON, 2012; DE NIES *et al.*, 2013, p. 2; FREIRE *et al.*, 2008; KOOP *et al.*, 2010b; MARINHO *et al.*, 2009, 2010, 2011a, 2011b; MATTOSO *et al.*, 2008). Then the references of this initial set of papers, whenever related to this work, were included (backward snowballing). Additionally papers that references this initial set of papers, whenever related to this work, were also included (forward snowballing). The process was repeated recursively with the new set of papers. To identify the references two search-engines were used: Scopus and Google Scholar.

It was considered related to this works any approach, product, or paper that deals with at least one of the secondary questions or the research question. It is important to highlight that ProvMonitor works with implicit provenance on files. It does not gather implicit provenance on other resources such as services, network (streaming), databases, etc. Therefore, the selected works were the ones related to implicit provenance on files. Some works that could potentially deal with implicit provenance on other environments were not included in this literature review. For example, the Wing/Pegasus system (KIM *et al.*, 2008)

can gather retrospective provenance that allows it to identify why the workflow trial was different from the specification. However, these differences are related to Pegasus running optimizations and not to implicit data flows effects. Approaches like that were not included. It is also important to highlight that ProvMonitor implicit provenance support is related to implicit data flows. There are other situations related to implicit provenance such as in the domain data. For example, SciCumulus (OLIVEIRA *et al.*, 2010) uses its *extractor* to track implicit values in the domain data. The works we review in this chapter were the ones related to implicit provenance of implicit data flows and not from other situations such as implicit values in the domain data. Work whose description and information was not publicly accessible or available on the databases accessible for us through CAPES, such as ACM Digital Library and IEEE Xplore, were also not included.

The remaining of this chapter presents the identified related work. Finally, at the end, Section 4.12 presents a comparison between the related works.

4.2 PASS

The Provenance-Aware Storage System (PASS) is a storage system that automatically collects, stores, manages, and provides search for provenance information (MUNISWAMY-REDDY *et al.*, 2006). It is a file and file-system oriented approach that collect provenance at the system level (operating system and file system), identifying operating system, library versions, and the environment information (*e.g.* environment variables) present on objects (*e.g.* files) creation. PASS automatically records command-line argument and the relationship between the various versions of the program and the performance results.

PASS collects provenance for every process, because it cannot know in advance, which processes might write to a PASS volume. Therefore, the amount of provenance gathered can be large, and not all of them directly related to the workflow trial, such as libraries and processes dependencies. A large amount of provenance information could be good to a system designer and to reproducibility; or bad (as distraction) to a user, such as a scientist.

PASS allows comparison of provenance of two pieces of derived data, such as simulation results, to reveal changes between two program invocations. It is capable of using provenance to identify the particular workflow that produced a document (provides a tight coupling between workflow management and information life-cycle management).

PASS tracks provenance at the file granularity. Data is considered to be either new data or the output of some process. Doing so, PASS is capable of gathering implicit

provenance. The provenance of a process output, on PASS, must include a unique reference to the particular instance of the executable (program) that created it; references to all input files; a complete description of the hardware platform on which the output was produced; and a complete description of the operating system and system libraries that produced the output. Additionally, it also includes the command line (OS command that invokes the program); the process environment; parameters to the process (frequently encapsulated in the command line or input files); and other data necessary to make pseudo-random computation repeatable.

PASS defines provenance that the system never sees as *opaque provenance*. This provenance is related to data from a non-PASS source, such as user, other computer, or other file system that is not provenance-aware (PASS has PASTA as its internal file system, which was developed using FiST, a toolkit for Linux to layer PASTA on top of any conventional file system). ProvMonitor also suffers from the limitation of data outside its management. Although, by relying on a VCS, ProvMonitor is more flexible on using different file systems. Indeed, by using VCS, ProvMonitor turns not provenance-aware file systems into provenance-aware ones.

It is important to highlight that PASS do not run on a VCS and besides retaining provenance information of old versions of files, it does not retain the old versions of files themselves (file content). Keeping provenance information of old versions of files is important to identify implicit provenance, however, losing these old versions of files contents make it harder to understand the effect of implicit provenance, unless the gathered provenance includes the description of what changed in the file. So, that way, by using the last version of the file and "reversing" the gathered changes, the old version of the file could be restored.

Another important point is that PASS maintains provenance information in memory and on disk. However, the two representations do not map one-to-one, since in memory PASS must account for processes and other objects that are not materialized in the file system (such as pipes and sockets). PASS does not track explicit data flow within a process, so all data a process accesses are gathered, since it can potentially affect the process' outputs. This strategy makes PASS aware of implicit provenance, however may significantly increase the amount of gathered provenance information.

PASS presents provenance management as a task for storage systems. Besides the improved provenance gathering capabilities by gathering provenance on storage systems, this gathering by itself can lead to a huge amount of provenance data, making it hard to scientist analysis. The analysis difficulty is hardened by the inexistence of a correlation between provenance and an abstract representation of the experiment such as activities of a workflow.

Summarizing, although capable of gathering implicit provenance, PASS has as main disadvantages the file system dependency, the large amount of captured provenance as a scientist distraction, provenance gathered that is not related to a more "user-friendly" abstract representation of the workflow (like on SWfMS), and the loss of old versions of files. ProvMonitor restrict the amount of provenance gathered to the scope of a workspace during a trial, relates the provenance gathered with workflow activities (abstraction representation defined by the scientist on the SWfMS), do file versioning, not losing old versions of files, and relies on a VCS. Although ProvMonitor has a VCS dependency, such dependency is less "environment-coupled" than a file system dependency.

4.3 ES3

The Earth System Science Server (ES3) project (FREW; METZGER; SLAUGHTER, 2008) focuses on Earth science data products derived from satellite remote sensing. It provides a local computing infrastructure to the scientist supporting exploratory computational science and operational product generation and dissemination. Provenance gathering on ES3 occurs at *science process* (workflow) execution time on the OS level. Thus, ES3 is able to gather provenance from any analysis software the scientist happens to be using. It automatically extracts provenance information by monitoring applications interactions (arguments, file I/O, system calls, and others) with the environment. The gathered information is then logged (*post hoc*) to the ES3 database, which stores the information as provenance graphs, represented in XML.

ES3 may use three different gathering strategies individually or together: *passive monitoring*, *overriding*, and *instrumentation*. The *passive monitoring* strategy traces the processes interactions with its environment. It does not involve any modifications to the process or environment. The *overriding* strategy replaces portions of the execution environment (such as shared libraries) with instrumented ones to explicit gather provenance information. It does not modify the *science process* itself, but requires detailed knowledge and access to the execution environment. Finally, the *instrumentation* strategy inserts specific instructions for provenance gathering into the *science process*. This strategy needs knowledge and access to the *science process*. Despite being more intrusive, this last strategy is more selective than the others are.

ES3 manages provenance through two components. The *probulator*, which is designed to monitor the execution of scientific applications non-intrusively, and the *ES3 core*, which decomposes the execution reports into object references and linkages between objects. The

ES3 core is capable of reconstructing the provenance graph at arbitrary starting points in time (forward and backward).

The main advantages of ES3 are to be SWfMS-independent, to be able to gather implicit provenance through OS, and to be capable of more selective gathering (through more intrusive strategies) when compared to gathering all OS system calls. However, ES3 has no notion of explicit workflow state and does not capture prospective provenance. Therefore, all gathered retrospective provenance is not related to any kind of abstract workflow representation. Finally, despite being more selective, gathering provenance at the OS level through system calls can lead to a large amount of provenance information, which may be prohibitive depending on the complexity of the analysis.

4.4 ASTRO-WISE

ASTRO-WISE (MWEBAZE; BOXHOORN; VALENTIJN, 2009) is a framework for data processing and data lineage for astronomical applications. It works by loading data into a database and making the database an integral part of all processing. With ASTRO-WISE, data can only be manipulated through interaction with the database. Access to the database is provided through persistence classes implemented on Python. The database has a schema (object attributes and method definitions) that can be extended through inheritance and polymorphism. To support the new persistent data products that may be created by the user, ASTRO-WISE allows the definition and addition of new processing routines. Therefore, users can modify functionality in modules, insert them into the system, or add modules on top of the already created ones, since these modules obey the standard data model.

The Python source code (that could be seen as the experiment specification) is stored in a VCS. However, a VCS cannot distinguish between method and classes in the source code, and it is the connection between classes and created objects that is tracked. Therefore, ASTRO-WISE assigns versions to classes, through an attribute, and uses class descriptors to track version changes, storing everything with the object (object versioning). ASTRO-WISE has been designed to compare versions of data and classes. If an object is identified as outdated, a request is triggered to compute the object “on-the-fly”.

Forcing all data accesses to be through the database, and by versioning data, ASTRO-WISE is capable of gathering implicit provenance. However, provenance gathered (versioned on the database) is related to the persistence class (versioned on a VCS with metadata on the database). Thus, to relate the gathered provenance with a more abstract representation of the workflow, users have to extend the persistence classes. Such process may require high

knowledge about ASTRO-WISE model and persistence classes' structure, which may be hard for a user such as a scientist. ASTRO-WISE uses a VCS only to versioning the persistence classes. With this strategy, ASTRO-WISE uses a VCS only as storage to script versioning, since provenance is gathered through other mechanisms. This use of VCS is quite different from the ProvMonitor's use, which employs a VCS to gather provenance during the workflow trial and is aware of the workflow behavior, going beyond a storage system.

4.5 STRONG LINKS

The Strong links approach (KOOP *et al.*, 2010b) posits that a "*tighter integration between scientific workflows and file management is necessary to enable the systematic maintenance of data provenance*". The approach presents a framework that couples provenance with the versioning of data produced and consumed by the workflow. Then, it captures the actual changes to data and detailed information about these changes.

They discuss that VCS can track changes and determine which changes effectively occurred, but are not capable of identifying how these changes occurred. On the other hand, provenance-enabled workflow systems are able to capture how changes occurred. However, they do not provide a systematic mechanism for maintain data, *e.g.*, given a file, they may not be capable of determining the workflow instance that generated it.

This discussion goes to the same direction of the concepts behind the ProvMonitor approach. Indeed, in software engineering, more specifically in the CM area, a VCS tracks what changed but not why a change occurred. To understand the "why", CM relies on issue tracking systems. Issue tracking systems are systems that describe issues that motivated a change. Therefore, to be capable of answering what changed and why a changed occurred, it is necessary to relate VCS gathered changes with the issues that motivated the change.

At the provenance scenario, it is the workflow specification that describes why and how a changed occurred. Thus, analogously, to be capable of answering what changed and why a changed occurred, it is necessary to relate the retrospective provenance (gathered through a VCS, for example) with the prospective provenance. This understanding guides the main concepts behind ProvMonitor.

In this same direction, the Strong Links approach uses a VCS to create strong links between the experiment and its provenance data. It is able to capture referenced file content, but it does not capture implicit provenance when workflow specification does not explicitly references the files as parameters. Relying on Git hash mechanism to identify changes on file contents, Strong Links adopts a cache mechanism to optimize workflow trials. ProvMonitor

also uses Git for storing file content separated from the workspace structure, optimizing persistence and being able to detect file moves. The main difference is that ProvMonitor is able to capture retrospective provenance even when it is not specified in the prospective provenance (that is, ProvMonitor captures implicit provenance). However, ProvMonitor does not present a cache mechanism. Cache mechanisms are a great improvement on deterministic scenarios, but on scenarios with implicit data flow it is risky to rely on a cache mechanism (that supposes some level of determinism). Thus, it is a tradeoff to rely on cache mechanism optimizations or to be implicit provenance aware.

4.6 CDE

CDE (GUO; ENGLER, 2011) is a system that monitors program execution of x86-Linux programs using *ptrace.* It packages the code, data, and environment required to run them on other x86-Linux machines. The creation of CDE package is completely automatic. Running programs within a package requires no installation, configuration, or root permissions. CDE is intended to be used in both academia and industry. However, there is no guarantee that all the dependencies required to run a package will be found. Therefore, it is up to the user to insert additional files into the package if necessary.

Using Linux *ptrace* to monitor the target program's system calls and to copy all of its accessed files, CDE is capable of gathering files implicitly touched during the program execution. However, CDE focus is on reproducibility. It does not present a provenance model, and do not relate the gathered information to any kind of representation of a workflow. Even more, by gathering all system calls it can gather OS and libraries dependencies of computer resources not related to the specialized system being executed (workflow). This is useful for reproducibility but not necessarily for scientist analysis. This is different of ProvMonitor target, which does not focus on packing the experiment. Instead, ProvMonitor focus on identifying implicit data flows and gathering its provenance, relating the gathered provenance with a workflow representation in a way that helps scientists' perception and understanding about implicit data flow occurrence and effects on the trial.

4.7 BURRITO

BURRITO (GUO; SELTZER, 2012) is a linux-based system that automatically captures researcher's computational activities during scripts-based experiments development and execution. It provides user interfaces to annotate the captured provenance and then allows making queries such as "Which script versions and command-line parameters generated the

output graph that this note refers to?" BURRITO consists of two parts: an extensible platform that automatically captures provenance and user activity context and a set of applications that allow the user to annotate and query the captured metadata stream.

BURRITO platform consists of a set of plugins that gather user activities on the user machine and integrates the gathered information on a MongoDB database. It uses a versioning file system (NILFS (KONISHI *et al.*, 2006)) for versioning of source code and data files. BURRITO also works at the OS level, gathering OS process access to files (I/O). Additionally it has a graphic user interface (GUI) trace daemon to gather user context information on GUI interactions such as "which application windows the user is viewing at all times while working on the experiments?" Through plug-in, BURRITO can capture user activities within specific applications.

As BURRITO gathers provenance during the workflow (script) development and execution, it might gather a high amount of provenance information not directly related to the trial. Additionally, as it works with scripts, it does not relate the gathered provenance with a user "abstract level of representation" of the experiment, such as workflow activities in a SWfMS. However, such limitation could be softened through relating gathered provenance with user-defined functions in the scripts. Names of user-defined functions may be closer to an abstraction representation of the workflow than other generic functions used in a script, even not being as closer as a workflow activity in a SWfMS might be.

ProvMonitor focuses on gathering provenance during the workflow trial and relates gathered provenance (retrospective) on the OS level (file system) with the correspondent workflow activity (prospective). Doing so, ProvMonitor relates provenance and file contents with the semantics used by the scientist during the workflow creation (workflows activities abstraction). Although ProvMonitor may imposes a high storage overhead during a trial, at the end ProvMonitor tends to store fewer amounts of provenance and more closed related to workflows activities execution than all process (OS) calls on user machine such as what is done by BURRITO.

4.8 SUMATRA

Sumatra (DAVISON, 2012) presents a set of best practices to simplify reproducibility. Among these practices are consistent and repeatable environment, version control, and separate code from configuration. Sumatra is a Python library for creating reproducible workflows. Operating with script-based workflows, it focuses on reproducibility and works by versioning the workflow specification and gathering environment information (such as

hardware configuration, operating system information, source code, among others), inputs, and outputs of the trial.

Sumatra uses a VCS for tracking and versioning the workflow specification and stores the outputs and parameters using a database. This is different from the main behavior of ProvMonitor, which does not focus on versioning of the workflow specification. Instead, ProvMonitor focus on following the workflow trial, gathering the changes on files produced by data flows during the execution. Thus, ProvMonitor uses VCS as part of the mechanism to support the workflow trial. Somehow, ProvMonitor operates as a complement of the SWfMS engine.

Sumatra is composed by two interfaces: a command-line, used to capture the context, input, and output of the workflow, and a web browser-based interface for viewing, searching, and annotating provenance. Although it uses a VCS, it does not relate the captured implicit provenance (using file versions) with prospective provenance data.

4.9 REPROZIP

ReproZip (CHIRIGATI; SHASHA; FREIRE, 2013) is an approach similar to CDE. It tracks OS calls from script-based workflows (command-line) and creates a package that contains all the binaries, files, and dependencies required to execute the workflow. Then, from the gathered retrospective provenance, ReproZip infers a workflow specification (it is not the prospective provenance) to prepare a package to be executed on Vistrails. After that, it is possible to extract the package on another environment, allowing the execution of the workflow in a new environment.

ReproZip only gathers retrospective provenance. Although the zip file contains part of the implicit provenance (it does not consider versions of files), VisTrails is not aware of this fact. Again, the focus of the approach is on packaging for reproducibility, while ProvMonitor focus is on understanding of the workflow trial and on identifying the effects of each workflow activity execution. Additionally, ignoring prospective provenance compromises the perception of the effects of implicit data flows, since without knowing what the workflow was supposed to do, it is not possible to identify a different behavior. Freire et al. (2008) also discuss the importance of gathering both prospective and retrospective provenance.

4.10 NOWORKFLOW

Not Only Workflow (noWorkflow) (MURTA *et al.*, 2014) is an approach to gather provenance on experiments executed as scripts. It is a command line tool, which transparently

captures provenance of scripts, including control flow information and library dependencies. NoWorkflow is non-intrusive and relies on Software Engineering techniques, including abstract syntax tree analysis, reflection, and profiling, to collect different types of provenance. The approach was developed for Python, a language with significant adoption by the scientific community. However, the ideas presented by the approach are language-independent and can be applied to other scripting languages.

The Main difference of noWorkflow approach compared to other script approaches is to be non-intrusive (it does not require users to change the way they work) and capable of gathering the equivalent of prospective and retrospective provenance (respectively definition and execution provenances). In fact, noWorkflow gather three different types of provenance: definition provenance, deployment provenance, and execution provenance. Definition provenance captures the structure of the script. It is equivalent to the prospective provenance. Deployment provenance captures the execution environment information. Finally, execution provenance captures the execution log for the script. It is equivalent to the retrospective provenance.

To gather execution provenance, noWorkflow implements specific methods of the Python profiling API and registers itself as a listener. During an execution, the profiler notifies the tool of all function activations in the source code. Such strategy may lead to a huge amount of provenance information. To minimize that, noWorkflow only registers function activations related to user-defined functions. Additionally, to be able of gather file access, noWorkflow overwrites the system's *open* function, altering its behavior to capture the content of the file, store it, call the original *open* system call, and then capture and store the file's content again. Indeed, noWorkflow waits the thread until the *open* function is about to end its scope, before gathering the file content. Therefore, noWorkflow waits by the file changes done by the *open* function. This allows noWorkflow to identify changes on files, including on implicit data flows, and relate the change with the function that triggered the change. It is similar to what is gathered by ProvMonitor. However, ProvMonitor works with a SWfMS and relates the changes on files with the correspondent workflow activities, which is more related to an annotation or abstraction representation of a workflow step than the name of a function in a script.

To store the gathered provenance, noWorkflow includes an embedded storage mechanism that does not require any installation or configuration. It is composed by a relational database and a database for file contents. Provenance on the two databases is related through the SHA1 hash codes of the content of files.

The main difference between noWorkflow and ProvMonitor is that noWorkflow operates over scripts. While ProvMonitor can also operate with scripts, its focus is to operate coupled with a SWfMS. Additionally, ProvMonitor relies on a VCS to manage the workspace. By relying on an overwrite of a system function, noWorkflow is capable of gather all access to files, thus, gathering provenance with a finer grain than the activity granularity used by ProvMonitor. However, gathering all file access may lead to information of files accesses not related to the workflow trial, thus, increasing the amount of "non-useful" provenance gathered. Additionally, by relying on workspaces ProvMonitor relies on and provides isolation to the activities/trials executions, preventing conflicts between concurrent activities/trials trying to access or change the same versions of files. Finally, relying on VCS opens opportunities to use specialized diff algorithms with ProvMonitor to analyze the history of changes on files on intra-trial and inter-trial analysis.

4.11 PROB

PROB (KOROLEV; JOSHI, 2014) is a system for ensuring provenance and reproducibility of Big Data workflows, with the MapReduce model, considering access to strictly controlled data. It uses a Git extension (Git-annex) to deal with very large files. Git-annex stores only hashes of datasets on the main repository and keep the content on a separated repository. It allows sharing only the hashes of the datasets without sharing datasets content itself. This is important on some scenarios, because some datasets of genetic information related to human subjects has strict access control through NCBI and institutional IRBs. Therefore, by sharing only files hashes, it is possible to guarantee that the scientists have the desired datasets even without sharing the content. Then scientists need to obtain datasets directly from the source authority through the appropriate protocols and directly inject it into their repository. This allows the workflow sharing without violating the terms of dataset usage.

PROB uses Git2Prov (DE NIES *et al.*, 2013) to extract and store provenance information from Git repository into the provenance database. The provenance database for the workflow description (built upon PIG (GATES *et al.*, 2009)) (prospective provenance) follows the PROV W3C model (LUC MOREAU; PAOLO MISSIER, 2012). This way PROB can gather prospective and retrospective provenance information and store them together, relating them.

PROB was developed to work with PIG. PIG is an environment to MapReduce workflows that uses Hadoop (APACHE, 2013), an open-source MapReduce implementation.

PIG has a specific language (PIG-latin (OLSTON *et al.*, 2008)) to allow the creation of MapReduce workflows in a higher abstraction level instead of manipulating directly the MapReduce implementations.

Before starting the workflow trial, PROB needs all nodes with Git-annex and the datasets repositories configured with references to each other. Manually configuring repositories on each node, referencing each other through remote repositories reference settings, may impose a high overhead to scientists. At the beginning and end of each *step* (workflow activity), the repositories states are gathered on each node involved in the *step*.

The usage of PIG (or MapReduce workflows) may demand more programming knowledge by the scientist then using the abstraction provided by SWfMS (that theoretically facilitates workflow development by scientists, since they do not demand improved programming skills). On PROB, the scientist needs to know, explicitly, they are using Git (specifically Git-annex extension) since the repositories need to be configured and datasets contents need to be injected into the repositories on each node. However, the paper that presents PROB does not present a more deeply discussion about how each branch is manipulated with Git. This may be a problem, since if it uses implicit branching on Git, when a repository is pushed to other repositories, Git may reject the push to prevent branches with multiple heads. Additionally, the MapReduce parallel data flows may be reduced and combined into a single data flow. The paper does not describe any details related to these *merge* situations and what happens with the provenance trace (versioned datasets traces) in such circumstances.

Although built to work on Big Data MapReduce workflows using PIG (upon Hadoop), the ideas behind PROB are generic and may be re-used on other scientific workflows tools. Indeed, compared with ProvMonitor, the ideas of PROB may be seen as another isolation strategy that presents interesting results to MapReduce workflows, especially with datasets access restrictions.

Our work, besides presenting a prototype that uses the ProvMonitor approach, also presents a more generic discussion about isolation strategies for workflows trials. The ProvMonitor approach proposes to look into the provenance issues through a CM perspective. Therefore, isolation strategies are one of the parts of this perspective related to provenance gathering. However, there are other aspects related to this provenance perspective through CM such as: to keep data traces; to manage data versions and to relate each version with an abstraction representation of the workflow created by the scientist (workflow activities on the SWfMS perspective); and to answer not only what changed, but also, why it changed.

Using Git, PROB can work on provenance gathering, on local dataset management (since the files are only references to remote repositories), and even on implicit provenance gathering over not explicitly specified accesses to the datasets in the local repositories. However, PROB relates the gathered provenance with the abstraction provided by PIG, which is closely related to scripts. Although it is possible to adapt PROB, there is no mention about dealing with abstractions used by scientists during workflow creation, such as the activities representation on workflows developed using SWfMS.

Concluding, the main difference from ProvMonitor is that ProvMonitor was designed to be more generic (not exclusive to MapReduce workflows), and being used coupled to SWfMS (*e.g.*, Vistrails and SciCumulus) or other provenance systems (*e.g.*, ProvManager). ProvMonitor focus is to gather implicit provenance on workflows developed on SWfMS. Finally, as mentioned, PROB strategy might be seen as another specific isolation strategy with interesting results on Map-reduce workflows with strictly controlled datasets accesses. However, the main idea of the ProvMonitor approach is to set the needed repositories automatically, not demanding manual configuration of each repository.

4.12 RELATED WORK COMPARISON

The identified approaches presented some general behaviors related to the research and secondary questions. To compare these behaviors, some dimensions were observed with some possible values. These dimensions are described below, and a comparison of the related work through these dimensions is presented in Table 8.

- **Implicit provenance aware/gathering:** Indicates if the approach can gather implicit provenance or gather provenance of implicit data flows. Possible values are “Yes” or “No”;
- **Intermediate data versioning:** Indicates if the approach gathers intermediate data and if it keeps all versions gathered, even when versioning is not explicitly used. Possible values are “Yes” or “No”;
- **Relates Prospective and Retrospective provenances:** Indicates if the approach relates prospective and retrospective provenance. Possible values are “Yes” or “No”;
- **VCS uses:** Indicates how the approach uses a VCS. Possible values are “None” if the approach does not use a VCS; “Source control” for the approaches that uses VCS for versioning the source code of the experiment; “Data” for approaches that uses VCS for versioning data files; “Workspace control” for approaches that uses VCS to control a specific workspace;

- **Gather mechanism (level):** Indicates the provenance gathering mechanism used by the approach. Possible values are “OS” for approaches relying on OS capabilities to gather provenance; “OS through libraries (such as Python libraries)” for approaches relying on OS capabilities, but using libraries to access OS gathering resources; “Workflow” for approaches that gather provenance on the workflow level, *e.g.* through a SWfMS; “Database” to approaches relying on database storage capabilities and resources monitoring (*e.g.*, services monitoring) to gather provenance; “VCS” for approaches relying on VCS gathering capabilities; “Versioning file system” for approaches relying on file systems with versioning capabilities to gather provenance;
- **Experiment representation:** Indicates the kind of workflow representation that the approach deals with. Possible values are “None” for approaches that completely ignore the scientific workflow concept; “OS process” for approaches that represent an execution based on the OS resources abstraction such as process and service IDs; “Script” for approaches whose workflow representation is based on scripts, using methods and classes as notation; “Workflow” for approaches that deal with the abstract representation of workflows such as what is done by SWfMS; “Generated Workflow” as the same as “Workflow”, but to approaches that references not the original workflow but a self-made one, calculated somehow;
- **Gathering moment:** Indicates when provenance is gathered. It is important information, because it provides hints about the amount of provenance collected and the moment the provenance gathered is related. For example, provenance gathered during workflow development my help analysis about workflow evolution, provenance gathered all time may help analysis about environment, and provenance gathered during the workflow trial may help analysis about data flows and workflows results. Possible values are “All time” for approaches that gather provenance all time, about everything happening on the environment; “Execution” for approaches that gather provenance during the trial or when the workflow is just about to start the trial; “Development” for approaches that gather provenance during the workflow development; “Workflow instrumentation” for approaches that gather provenance during workflow instrumentations/modifications, after workflow development and before workflow running;

Table 8: Related work comparison

Related Work	Comparison dimensions						
	Implicit provenance aware / gathering	Intermediate data versioning	Relates Prospective and Retrospective provenances	VCS uses	Gather mechanism (level)	Experiment representation	Gathering moment
PASS	Yes	No	No	None	OS	OS Process	All time
ES3	Yes	No	No	None	OS	OS Process	Execution
ASTRO-WISE	Yes	Yes	Yes	Persistence classes track	Database	Script	Execution
STRONGLINKS	No	Yes	Yes	Data	VCS	Workflow	Execution
CDE	Yes	Yes	No	None	OS	None	Execution
BURRITO	Yes	Yes	Yes	none	OS and Versioning file system	Script / OS Process	All time
SUMATRA	Yes	Yes	No	Source Code	VCS / OS through Python Libraries	Script	Development and Execution
REPROZIP	Yes	No	No	None	OS through Python Libraries	Script / Generated Workflow	Execution
NOWORKFLOW	Yes	Yes	Yes	None	Python Libraries and OS	Script	Execution
PROB	Yes	Yes	Yes	Workspace control	VCS	Script	Execution
PROVMONITOR	Yes	Yes	Yes	Workspace control	VCS / OS through file system metadata	Workflow	Workflow instrumentation and Execution

4.13 FINAL REMARKS

The main difference between these approaches and ProvMonitor is that ProvMonitor works on both workflow and OS domains, being capable of associate provenance captured at OS level with the workflow activities that produced them. Additionally ProvMonitor uses VCS to follow the workflow trial, not only as a storage system or for workflow versioning,

and uses configuration management techniques (such as VCS capabilities) to help on the gathering, storage, and analysis of such data. Thus, it helps scientists to understand data transformations along the trial.

The exception is the PROB approach, which is the closer related to the ProvMonitor approach. As discussed, ProvMonitor presents a more generic discussion about isolation strategies, focusing on gathering implicit provenance on workflows developed using SWfMS workflows representations, and designed to be coupled with SWfMS or provenance systems. Therefore, PROB strategy may be seen as another isolation strategy, specific to MapReduce workflows with strictly controlled dataset accesses.

The next Chapter presents the conclusion of this work.

CHAPTER 5 – CONCLUSION

5.1 CONTRIBUTIONS

Provenance has still many open issues to be tackled, especially those related to intermediate data versioning and implicit data flow provenance. The main contribution of this work is the introduction of a new approach to gather implicit provenance while versioning intermediate data, through a CM perspective, entitled ProvMonitor. The CM perspective helped to identify the need of isolation on the gathering of provenance during a workflow trial. This approach works as a hybrid approach, operating on both workflow and OS domains considering context isolation (workflow through workspaces and version history through branches), gathering provenance related to implicit data flow and relating the gathered provenance with the abstract representation of the workflow (workflow activities) that produced such provenance. The concepts of the approach are general and could be applied on different SWfMS. Indeed, ProvMonitor successfully added implicit provenance awareness on SciCumlus SWfMS and Vistrails.

The approach was evaluated by a case study on a real scientific workflow (SciPhy). An expert researcher (pharmaceutical scientist) supported the evaluation. The objective was to evaluate the effectiveness and the efficiency of the approach, evaluating the benefits of gathering implicit provenance on the workflow analysis and results understanding. The effectiveness evaluation targeted on evaluating the approach gathering capabilities, query capabilities over the gathered information, and how the scientist could benefit from querying the gathered information. The efficiency evaluation targeted to evaluate the overhead imposed by the approach on a workflow trial over two different dimensions: execution time and storage consumption.

Experimental results have shown the effectiveness of the approach on the implicit provenance gathering by considering context isolation and relating provenance information gathered in each domain. This effectiveness, in addition with related work that also use VCS to support provenance gathering and storage, shows that CM presents an interesting perspective to deal with the provenance issues. Indeed, the understanding about the several presented strategies shows that a provenance gathering strategy that is aware of the workflow trial behavior can contribute more than just capturing provenance. This is another contribution of this work: to apply a CM perspective on the gathering of provenance information during

workflow trials, instead of using CM tools only as storage of data or to support the workflow development steps, such as versioning the workflow specification.

Thus, the strategy we proposed allows the gathering of provenance information even on implicit data flows. Additionally, by manipulating data on a controlled way, it is also possible to achieve data management capabilities, such as data versioning, historical data storage control, support for data transfer, and more. By managing both data and provenance it is possible to achieve a tighter coupling between data and its provenance when compared to approaches that manage provenance and data separately (MUNISWAMY-REDDY *et al.*, 2006).

Finally, relating prospective and retrospective provenance, while controlling implicit provenance, opens some interesting analysis possibilities inspired on CM. It is possible to validate a workflow by verifying: if everything that was specified is being executed and if everything that is being executed was specified, or, at least, if it is being observed and analyzed. Such perception allows the scientist to identify the influence of implicit data flows or even the influence of intermediate data over the results. Yet, under the CM perspective, it is also possible to perform intra-trial and inter-trial analysis. The first one compares changes produced by two different activities at the same trial. The second compares changes produced by the same activity but in different trials. All of this grants to the scientists opportunity to improve their own understanding about the trial, observing the impact of implicit data flows and parameters changes on each workflow activity and on the workflow results.

5.2 LIMITATIONS

Our experiments also highlighted the overhead imposed during workflow trials, evidencing that some gathering strategies can be prohibitive in some situations. Although we are convinced of the benefits of using VCS on the provenance gathering and storage processes, results has shown that ProvMonitor has to be more carefully tailored to the SWfMS due to the imposed overhead. Part of this overhead is due to the chosen gathering strategy. However, other factors may influence the results. For example, identification of file accesses is done by accessing and reading each file metadata. Although ProvMonitor tries to optimize it by accessing files only inside accessed directories, some accessed directories may present a large number of files to be verified. Thus, a more efficient way of identifying file accesses should be explored.

To minimize the overhead imposed by ProvMonitor, some Git optimizations mechanisms may be explored. Git provides support to hard-links (over file systems which

support them) to avoid unnecessary data replication. Through hard links, a copy of the content of a file is done only when some change occurs, since at this moment different references to the same file start to point to different files contents. Other interesting mechanism provided by Git is repository sharing. Through this resource, two different workspaces may share the same repository (content objects). However, repository sharing has to be used only locally (workspaces at the same disk/storage). Both hard-links and repository sharing may provide two benefits: avoid data replication and speed up the workspace setup.

To minimize the overhead imposed by identifying files access, instead of sequentially verifying accesses to files after each activity, a listener, tied to the operating system, could be used to monitor the workspace. The listener would react to OS notifications of accesses on workspace files in the moment of the accesses, thus eliminating the need of scanning the entire workspace after each activity, avoiding overhead of execution time spent on workspace scanning. Indeed, Java 7 provides an API called WatchService (ORACLE, 2014) that could be explored. This API offers a listener that operates over the Java Virtual Machine (JVM) providing abstraction about the file system and the OS.

It is important to highlight that ProvMonitor works at the scope of a workspace. Thus, it is important to guarantee that the experiment will work only inside the workspace. This could be achieved by coupling the SWfMS together with ProvMonitor, as we did with SciCumulus in this work. Another option is using a workflow isolation strategy, setting the workspace on a root directory, but this can decrease the quality of provenance gathered by sacrificing isolation. Finally, it is also possible to use OS resources such as the "*chroot*" command that redirect access from a directory to another, allowing redirecting file system accesses to the desired workspace.

Another limitation of the approach is related to its gathering granularity. ProvMonitor works by gathering provenance considering the granularity of a workflow activity. Through a CM perspective, it seems reasonable to consider activity as a milestone of a workflow computation completion, since an activity represents an abstract annotation (done by the scientist that specified the workflow) about a workflow computation/step. Thus, gathering provenance about an activity seems to present an interesting grain, not too fine neither too coarse. However, some workflow activities may execute more than one computation, presenting its own "internal" implicit data flow. By gathering provenance at the activity granularity, only the result of an activity execution is gathered. The activity "internal" implicit data flow is lost. Thus, it could be interesting to achieve a finer grain of provenance gathering in some situations like debugging an activity. Again, the use of a listener could benefit of OS

notifications to react on changes into the workspace, triggering VCS commits at the workspace in the moment that the changes occur, not only at the end of an activity execution. Thus, it could allow provenance gathering during the workflow activity execution, not just at the end of it. This could be a strategy to achieve finer grain provenance gathering (other than the workflow activity).

Another benefit of a listener is to explore OS monitoring capabilities instead of building upon the application layer. Monitoring the workspace with a listener, allows the system to react on every change into the workspace, for example, by gathering the changed file and storing it on a separated repository. This strategy is similar to what is done by the Strong Links approach (KOOP *et al.*, 2010b). The difference is that Strong Links operates only with what is defined in the workflow specification, while the listener operates at the OS domain and is aware of implicit data flows in the managed workspace. Additionally, isolating the repository from the workspace would avoid data replication/redundancy. This could also be achieved through a central VCS (*e.g.*, SVN) instead of a distributed VCS (*e.g.*, Git) or cloning with the Git share capabilities. However, relying on a listener instead of a VCS to manage the workspace improves the flexibility of the workspace setup and management at the price of the responsibility to support the benefits provided by a VCS. Thus, the gathering mechanism would need to manage the workspace, relying on some of the same foundations of VCS (isolation of workspace, isolation of history, and separating working copy of history/versions) instead of using a VCS to manage the workspace directly. This could also be improved by enriching the provenance model with some capabilities of some VCS (used as a central repository) to identify objects' similarities. Identifying similarities between files could provide hints about a file origin, transformations, and about file relationships.

Finally, the discussion about isolation and gathering strategies showed that following the workflow trial might lead to forks and merges situations related to data flows during a trial. However, the current version of ProvMonitor cannot deal with more complex merge situations. It is only capable of dealing with forks that do not merge back to a single data flow or to deal with concurrent data flows that go back to a single data flow through the selection of one data flow (in detriment of the others). Thus, support to data flows merge is a limitation of this work.

5.3 FUTURE WORK

We could identify some improvements and research possibilities to extend the provenance gathering capabilities and to better explore and query the gathered provenance. In

the remaining of this section, we describe some possible researches and improvements in the ProvMonitor approach.

ProvMonitor is a hybrid approach that relies on the workflow and OS domains. However, the action over the OS domain is restricted to files. Thus, an interesting research opportunity is to extend the implicit gathering capabilities beyond the files. For example, gathering implicit provenance over networks, streaming, or database accesses, but identifying ways of relating the gathered provenance with the workflow activities or workflow representation that produced the gathered provenance.

This work presented some different isolation strategies that could be used on different provenance gathering strategies. On the other hand, the PROB approach (KOROLEV; JOSHI, 2014) presented a different strategy that seems to be efficient to MapReduce workflows with data access restrictions. Yet there are other specific workflows situations that could benefit of specific isolation and gathering strategies. Thus, a research opportunity is to explore the concepts behind the ProvMonitor approach to develop and improve isolation and provenance gathering strategies to specific situations among specific workflows domains.

In this direction, following the trial behavior and data flow can lead to merge situations. As discussed, ProvMonitor merge capabilities are quite limited. Thus, a research opportunity that may present challenging obstacles is to identify strategies to provide support to automatic merge of data flows. Following the data flow and managing merge of data flows may evidence the critical path of data that effectively contributed to the workflow result and data that do not presented any kind of influence over the results. Such perception may be explored together with VCS pruning capabilities, as a strategy to control provenance explosion in the provenance repository (one of the open issues related to provenance management) discarding provenance of irrelevant data flows or not related to the workflow result ("junk provenance"). Alternatively, instead of discarding provenance, some provenance compression techniques may be developed.

Finally, after gathered, provenance must be queried and analyzed to be useful. Thus, a user-friendly query and exploratory mechanism must be developed to help scientists to explore data provenance in a useful fashion. Such query mechanism must be capable of querying over different provenance databases (*e.g.*, provenance in the relational databases or VCS repository), providing an abstraction of the persistence resources being used. It must allow the scientists to worry only about the workflow abstraction, querying about workflow activities and produced/consumed files, instead of different database implementations, specific VCS repository resources, and other specific implementation resources such as VCS

branches. Therefore, the resources of the gathering mechanism must be seen only as the plumbing of a “provenance suit”. Thus, it would allow the scientists to look at the workflows trials, results, and provenance over the abstraction of the workflow representation. In other words, it would allows scientist to look at workflows as files and executions/trials, instead of OS process, branches, repositories, and other “low level" resources. Indeed, such query mechanism may be coupled together with ProvMonitor and ProvManager, composing a provenance suit, independent of SWfMS, but capable of operating together with any SWfMS.

BIBLIOGRAPHY

ALTINTAS, I.; BARNEY, O.; JAEGER-FRANK, E. Provenance Collection Support in the Kepler Scientific Workflow System. In: MOREAU, L.; FOSTER, I. (Eds.). Provenance and Annotation of Data. Lecture Notes in Computer Science. [s.l.] Springer Berlin / Heidelberg, 2006. v. 4145.

APACHE. Hadoop: Open-source implementation of MapReduce, 2013. Available at: <http://hadoop.apache.org>. Accessed: dec. 7, 2014

ARAGON, C. R.; RUNGE, K. J. Workflow management for high volume supernova searchACM symposium on Applied Computing. Proceedings...ACM, 2009

BIVAR, B. et al. Uma Comparação entre os Modelos de Proveniência OPM e PROV. In: BRESCI - VII BRAZILIAN E-SCIENCE WORKSHOP. Maceió - Brazil: jul. 2013

BOUGANIM, L.; FLORESCU, D.; VALDURIEZ, P. Dynamic load balancing in hierarchical parallel database systems. 1996.

CALLAHAN, S. P. et al. VisTrails: visualization meets data managementACM SIGMOD. Proceedings... In: SIGMOD '06. ACM Press, 2006 Available at: <http://portal.acm.org/citation.cfm?doid=1142473.1142574>. Accessed: jun. 17, 2012

CHACON, S. Pro Git. 1. ed. Berkeley, CA, USA: Apress, 2009.

CHIRIGATI, F.; SHASHA, D.; FREIRE, J. Packing experiments for sharing and publicationACM SIGMOD. Proceedings...ACM, 2013

COBENA, G.; S. ABITEBOUL; MARIAN, A. Detecting Changes in XML DocumentsICDE. Proceedings... In: INTERNATIONAL CONFERENCE ON DATA ENGINEERING (ICDE). ICDE, 2002

CONRADI, R.; WESTFECHTEL, B. Version Models for Software Configuration Management. ACM Computing Surveys, v. 30, n. 2, p. 232–282, jun. 1998.

DA CRUZ, S. M. S. et al. Using explicit control processes in distributed workflows to gather provenance. In: Provenance and Annotation of Data and Processes. [s.l.] Springer, 2008.

DA CRUZ, S. M. S. et al. Capturing distributed provenance metadata from cloud-based scientific workflows. Journal of Information and Data Management, v. 2, n. 1, p. 43, 2011.

DANTAS, C. R.; MURTA, L. G. P.; WERNER, C. M. L. Mining Change Traces from Versioned UML RepositoriesBrazilian Symposium on Software Engineering (SBES). Proceedings...João Pessoa, Brazil: 2007

DART, S. Concepts in configuration management systemsACM Press, 1991 Available at: <http://portal.acm.org/citation.cfm?doid=111062.111063>. Accessed: mar. 18, 2012

DAVISON, A. P. Automated Capture of Experiment Context for Easier Reproducibility in Computational Research. Computing in Science Engineering, v. 14, n. 4, p. 48–56, 2012.

DE NIES, T. et al. Git2PROV: Exposing Version Control System Content as W3C PROVPosters & Demonstrations Track within the 12th International Semantic Web Conference (ISWC-2013). Proceedings...CEUR-WS, 2013

DEAN, J.; GHEMAWAT, S. MapReduce: simplified data processing on large clusters. Communications of the ACM, v. 51, n. 1, p. 107–113, 2008.

DEELMAN, E. et al. Workflows and e-Science: An overview of workflow system features and capabilities. Future Generation Computer Systems, v. 25, n. 5, p. 528–540, may 2009.

FELSENSTEIN, J. PHYLIP-phylogeny interference package (version 3.2). Cladistics, v. 5, p. 164–166, 1989.

FELSENSTEIN, J. PHYLIP (phylogeny inference package) Distributed by the author. Department of Genome Sciences, University of Washington, Seattle, 2005.

FREIRE, J. et al. Provenance for Computational Tasks: A Survey. Computing in Science & Engineering, v. 10, n. 3, may 2008.

FREW, J.; METZGER, D.; SLAUGHTER, P. Automatic capture and reconstruction of computational provenance. Concurrency and Computation: Practice and Experience, v. 20, n. 5, 2008.

GATES, A. F. et al. Building a high-level dataflow system on top of Map-Reduce: the Pig experience. Proceedings of the VLDB Endowment, v. 2, n. 2, p. 1414–1425, 2009.

GILBERT, D. Sequence File Format Conversion with Command-Line Readseq. Current Protocols in Bioinformatics, p. A–1E, 2003.

GONÇALVES, J. C. DE A. et al. Using domain-specific data to enhance scientific workflow steering queries. In: Provenance and Annotation of Data and Processes. [s.l.] Springer, 2012. p. 152–167.

GUO, P. J.; ENGLER, D. CDE: Using system call interposition to automatically create portable software packagesProceedings of the 2011 USENIX Annual Technical Conference. Proceedings...2011

GUO, P. J.; SELTZER, M. Burrito: wrapping your lab notebook in computational infrastructureProceedings of the 4th USENIX conference on Theory and Practice of Provenance, TaPP. Proceedings...2012

HEY, T., TANSLEY, STEWART; TOLLE, K. The fourth paradigm data-intensive scientific discovery. Redmond, Wash.: Microsoft Research, 2009.

HUNT, J. W.; MCILROY, M. D. An Algorithm for Differential File Comparison. Murrray Hill, NJ: Bell Laboratories, 1976.

KATOH, K.; TOH, H. Parallelization of the MAFFT multiple sequence alignment program. Bioinformatics, v. 26, n. 15, p. 1899–1900, 2010.

KEANE, T. M. et al. Assessment of methods for amino acid matrix selection and their use on empirical data shows that ad hoc assumptions for choice of matrix are not justified. BMC evolutionary biology, v. 6, n. 1, p. 29, 2006.

KIM, J. et al. Provenance trails in the wings/pegasus system. Concurrency and Computation: Practice and Experience, v. 20, n. 5, p. 587–597, 2008.

KONISHI, R. et al. The Linux implementation of a log-structured file system. ACM SIGOPS Operating Systems Review, v. 40, n. 3, p. 102–107, 2006.

KOOP, D. et al. Bridging workflow and data provenance using strong linksSSDBM'10 Proceedings of the 22nd international conference on Scientific and statistical database management. Proceedings... In: 22ND INTERNATIONAL CONFERENCE ON SCIENTIFIC AND STATISTICAL DATABASE MANAGEMENT. 2010a

KOOP, D. et al. Bridging workflow and data provenance using strong linksSSDBM. Proceedings... In: INTERNATIONAL CONFERENCE ON SCIENTIFIC AND

STATISTICAL DATABASE MANAGEMENT (SSDBM). Springer-Verlag, 2010b Available at: <http://dl.acm.org/citation.cfm?id=1876037.1876071>

KOROLEV, V.; JOSHI, A. PROB: A tool for Tracking Provenance and Reproducibility of Big Data Experiments. Reproduce'14. HPCA 2014, 2014.

LUC MOREAU; PAOLO MISSIER. PROV-DM: The PROV Data Model. W3C Working Draft. Available at: <http://www.w3.org/TR/prov-dm/>.

MARINHO, A. et al. A Strategy for Provenance Gathering in Distributed Scientific WorkflowsIEEE, jul. 2009 Available at: <http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5190676>. Accessed: mar. 18, 2012

MARINHO, A. et al. Managing provenance in scientific workflows with provmanagerInternational Workshop on Challenges in e-Science-SBAC. Proceedings...2010

MARINHO, A. et al. Challenges in managing implicit and abstract provenance data: experiences with ProvManagerTaPP. Proceedings... In: USENIX WORKSHOP ON THE THEORY AND PRACTICE OF PROVENANCE (TAPP). 2011a

MARINHO, A. et al. ProvManager: a provenance management system for scientific workflows. Concurrency and Computation: Practice and Experience, v. 24, n. 13, oct. 2011b.

MATTOSO, M. et al. Gerenciando Experimentos Científicos em Larga EscalaIn: SEMISH - CSBC. Proceedings... In: SEMISH – SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE. Belém, Pará - Brasil: 2008

MATTOSO, M. et al. Towards supporting the life cycle of large scale scientific experiments. International Journal of Business Process Integration and Management, v. 5, n. 1, p. 79 – 92, 2010.

MILES, S. First Provenance Challenge, 2006a. Available at: <http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge>

MILES, S. Second Provenance Challenge, 2006b. Available at: <http://twiki.ipaw.info/bin/view/Challenge/SecondProvenanceChallenge>

MOREAU, L. et al. The Open Provenance Model: An Overview. In: FREIRE, J.; KOOP, D.; MOREAU, L. (Eds.). Provenance and Annotation of Data and Processes. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. v. 5272p. 323–326.

MOUALLEM, P. et al. Tracking Files in the Kepler Provenance Framework. In: WINSLETT, M. (Ed.). Scientific and Statistical Database Management. Lecture Notes in Computer Science. [s.l.] Springer Berlin Heidelberg, 2009. p. 273–282.

MUNISWAMY-REDDY, K. et al. Provenance-aware storage systemsAnnual Conference on USENIX. Proceedings...USENIX Association, 2006

MURTA, L. et al. noWorkflow: Capturing and Analyzing Provenance of Scripts. In: INTERNATIONAL PROVENANCE AND ANNOTATION WORKSHOP (IPAW). 2014

MWEBAZE, J.; BOXHOORN, D.; VALENTIJN, E. Astro-wise: Tracing and using lineage for scientific data processingNetwork-Based Information Systems, 2009. NBIS'09. International Conference on. Proceedings...IEEE, 2009

OCAÑA, K. A. et al. SciPhy: a cloud-based workflow for phylogenetic analysis of drug targets in protozoan genomes. In: Advances in Bioinformatics and Computational Biology. [s.l.] Springer, 2011.

OCAÑA, K. A. et al. Designing a parallel cloud based comparative genomics workflow to improve phylogenetic analyses. Future Generation Computer Systems, v. 29, n. 8, p. 2205–2219, 2013.

OCAÑA, K. A. et al. Exploring Large Scale Receptor-Ligand Pairs in Molecular Docking Workflows in HPC Clouds. In: 13TH IEEE INTERNATIONAL WORKSHOP ON HIGH PERFORMANCE COMPUTATIONAL BIOLOGY (HICOMB 2014). Phoenix, Arizona, USA: IEEE, 2014

OGASAWARA, E. et al. An algebraic approach for data-centric scientific workflows. Proc. of VLDB Endowment, v. 4, n. 12, p. 1328–1339, 2011.

OHST, D.; WELLE, M.; KELTER, U. Differences between versions of UML diagrams. In: ESEC. ACM, 2003 Available at: <http://portal.acm.org/citation.cfm?id=940071.940102>. Accessed: jan. 11, 2009

OLIVEIRA, D. C. M. DE et al. Scicumulus: A lightweight cloud middleware to explore many task computing paradigm in scientific workflowsIEEE CLOUD. Proceedings...IEEE, 2010

OLIVEIRA, D. C. M. DE. Uma Abordagem De Apoio À Execução Paralela De Workflows Científicos Em Nuvens De Computadores. [s.l.] Universidade Federal do Rio de Janeiro, 2012.

OLSTON, C. et al. Pig latin: a not-so-foreign language for data processingProceedings of the 2008 ACM SIGMOD international conference on Management of data. Proceedings...ACM, 2008

ORACLE. Watching a Directory for Changes, 2014. Available at: <http://docs.oracle.com/javase/tutorial/essential/io/notification.html>. Accessed: jul. 6, 2014

PRUITT, K. D. et al. NCBI Reference Sequences: current status, policy and new initiatives. Nucleic acids research, v. 37, n. suppl 1, p. D32–D36, 2009.

ROCHKIND, M. J. The Source Code Control System. IEEE transactions on software engineering, v. 1, n. 4, 1975.

ROKAS, A. Phylogenetic analysis of protein sequence data using the Randomized Axelerated Maximum Likelihood (RAXML) Program. Current Protocols in Molecular Biology, p. 19–11, 2011.

SILVA JUNIOR, J. R. et al. A GPU-based Architecture for Parallel Image-aware Version ControlCSMR. Proceedings... In: EUROPEAN CONFERENCE ON SOFTWARE MAINTENANCE AND REENGINEERING (CSMR). Szeged: IEEE, mar. 2012

WALKER, E.; GUIANG, C. Challenges in Executing Large Parameter Sweep Studies Across Widely Distributed Computing Environments5th IEEE Workshop on Challenges of Large Applications in Distributed Environments. Proceedings...: CLADE ’07.ACM, 2007 Available at: <http://doi.acm.org/10.1145/1273404.1273411>