THE UNIVERSITY of ADELAIDE

An Adaptive Provenance Collection Architecture in Scientific Workflow Systems

Thesis submitted in accordance with the requirements of
the University of Adelaide for the degree of Doctor in Philosophy
by

Mehdi Sarikhani

Supervisors:
Dr. Andrew L. Wendelborn
Dr. Bradley Alexander

Faculty of Engineering, Computer and Mathematical Sciences (ECMS)
School of Computer Science
The university of Adelaide
April 2015

ABSTRACT

This thesis investigates adaptive provenance collection in the context of scientific workflow systems. In particular, we show how to design and implement an adaptive provenance system that operates at multiple levels of granularity.

Scientists in different disciplines use scientific workflows as management and representational frameworks for distributed scientific computations. Scientific workflow systems need a scientific workflow management system (SWfMS) to manage the flow of work among (both local and distributed) participants and resources; and to coordinate user and system participants. Scientific workflow systems are run over heterogeneous environments, which see changes over time in resources, requirement and policies (e.g. the cost of resources, or the policy of provenance collection in). Such changes may influence the way in which workflow mechanisms can best operate within the environments, and motivate our consideration of adaptive mechanisms to deal with such changes.

SWfMSs run a scientist's experiments. They manage sequences of complex transformational processes; in particular, they collect provenance information at various levels of abstraction (or granularity). Provenance in SWfMS is important because it enables scientists to have a clear understanding of results, especially to reproduce and verify them.

Provenance information can be collected at different levels of detail, typically coarse, medium and fine grained, using specific provenance collection mechanisms. We define a Model of Provenance (MoP) for each level to make it explicit what is determined as provenance information in each level, and in addition how it is represented.

We explore and survey provenance collection mechanisms and MoP, in order to provide sufficient understanding of the design and development of suitable provenance mechanisms for workflow systems. We emphasize adaptability and interoperability as important and desirable properties of a provenance system, especially those running over distributed environments.

We propose a novel provenance architecture in scientific workflow architectures, which benefit from the notion of *separation of concerns*, which is an important principle in middleware architecture. The design and development of our adaptive provenance architecture untangles the adaptive-granularity and provenance-collection concerns, so that we can more easily offer adaptive provenance collection mechanisms.

We use reflection (MetaObject Protocol (MOP)) and Aspect-Oriented Programming (AOP) as two ways of realizing the separation of concerns in our adaptive provenance collection mechanisms. Both the MOP and AOP oriented adaptive provenance collection mechanisms are explored in our scientific workflow case study, and implemented on a process network based workflow model. The case study demonstrates adaptive collection and representation of multiple levels of provenance granularity, according to our model of provenance (MoP). This MoP represents various levels of provenance granularity in a format compatible with a generic Open Provenance Model, enabling interoperability.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Advanced Message Queuing Protocol	AMQP	Simple Storage Service	S3
Aspect-Oriented Programming	AOP	Synchronous Dataflow	SDF
Aspect-Oriented Software Development	AOSD	Support for Provenance Auditing in Distributed Environments	SPADE
Collection-oriented modelling and design	COMAD	Simple Queuing Service	SQS
directed acyclic graph	DAG	SQL Server Data Services	SSDS
Dynamic Dataflow	DDF	Scientific Workflow Management System	SWfMS
Description-Driven System	DDS	Scientific Workflow Provenance Data Model	SWPDM
Earth System Science Server	ES3	Transparent Result Caching	TREC
Earth System Science Workbench	ESSW	Ultrascale Visualization Climate Data Analysis Tools	UV-CDAT
Geographic information System	GIS	VIsual sciEntific Workflow management system	VIEW
Intra-process Used	I-Used	WasControlledBy	WCB
Intra-process WasDerivedFrom	I-WDF	Windows Communication Foundation	WCF
Intra-process WasGeneratedBy	I-WGB	WasDerivedFrom	WDF
Intra-process WasIndirectlyInvokedBy	I-WIIB	Workflow Management Coalition	WfMC
Lineage File System	LinFS	workflow management system	WfMS
Language Independent Query	LINQ	WasGeneratedBy	WGB
Model of Computation	MoC	WasTriggeredBy	WTB
Modelling Markup Language in XML	MoML	Markup language	XML
Model of Provenance	MoP		
MetaObject Protocol	MOP		
neuGRID for You	N4U		
object-oriented programming	OOP		
Open Provenance Model	OPM		
Open Provenance Model for Workflows	OPMW		
Provenance-aware Storage System	PASS		
Process Networks	PN		
Process Network Application	PNA		
Provenance Interchange Language	PROV		
Quality of Services	QoS		

DECLARATION

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission for any other degree or diploma in any university or other tertiary institution without the prior approval of the University of Adelaide and where applicable, any partner institution responsible for the joint-award of this degree.

I give consent to this copy of my thesis, when deposited in the University Library, being made available for loan and photocopying, subject to the provisions of the Copyright Act 1968.

I also give permission for the digital version of my thesis to be made available on the web, via the University's digital research repository, the Library catalogue and also through web search engines, unless permission has been granted by the University to restrict access for a period of time.

Mehdi Sarikhani

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisors, Andrew and Brad, for their invaluable guidance and assistance with my work. Both have been excellent supervisors, and provided a great deal of insightful advice and feedback that has guided my project.

Many thanks go to my parents, who have done so much over the years to support and encourage me in my studies, and teach me the value of working hard to pursue my goals. To fully describe the ways in which they have helped me get to where I am today would take another three hundred pages. It is to them that I dedicate this thesis. Finally, to those always support me in every second of this journey in various ways -Maman joon, Azar, Mohammad, Ali, and Avin - thanks for making my life enjoyable.

1 INTRODUCTION

Hey *et al.* [1] identified four paradigms or stages in the history of Science (Figure 1.1). First, there was the empirical paradigm, where scientists merely observed and described natural phenomena. Next, the theoretical paradigm developed where general rules were applied to scientific phenomena. With the advent of computers and especially High Performance Computing in the last few decades, tremendous changes in the science paradigm and scientific achievements have occurred. In this third paradigm, computers make it possible for scientists to simulate complex theoretical models, which generate large quantities of data. Most recently, scientists have focused on using already collected experimental scientific data instead of actual instruments. For example, scientists use simulation data from the Hubble telescope instead of using the actual telescope in order to look at stars in sky [1]. This fourth paradigm has emerged because scientists have realized that it is possible to reuse previous data and results and that it is unnecessary to produce these data again. In addition, they have become increasingly aware that these massive amounts of data will be useless, unless they can be stored and processed appropriately.

The fourth paradigm is characterized by expressing the distinction between data-intensive science (such as, experimental and theoretical science) and computational science. This means that scientists in different disciplines face a data deluge [1] which needs a well-designed infrastructure to store the gathered experimental data and process them to solve scientific problems. Thus, many activities following from fourth paradigm benefit from High Performance Computing (HPC), in order to manage and analyse vast quantities of collected data. In the fourth paradigm, IT and other sciences conjoin and eScience has emerged. In order to solve scientists' problems in eScience, scientific workflow systems (as explained in section 1.2) [2-6] (with the help of Scientific Workflow Management Systems (SWfMS) [7-13]) have emerged to automate processes of collecting, storing, processing, analysing, visualizing and publishing data. In this situation, workflow systems can be considered as a problem solving environment that manages and represents complex, heterogeneous and distributed scientific computations. These computations may need to be executed, used or managed fully or partially on a variety of distributed resources [9, 11, 14, 15].

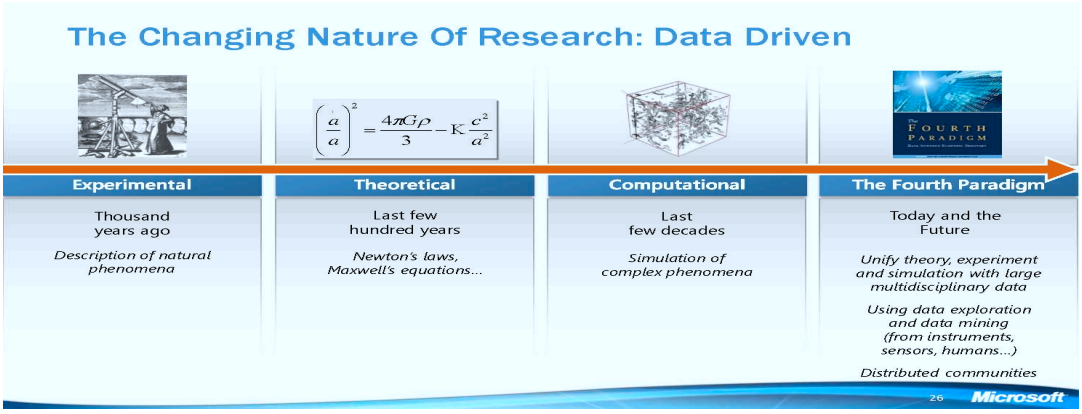


Figure 1.1.Science Paradigms [1].

According to Hey *et al.* [1], the following is a definition of scientific workflow: "a workflow is a precise description of a scientific procedure—a multi-step process to coordinate multiple tasks, acting like a sophisticated script". In this paradigm, scientific experiments reuse generated and shared raw datasets and reproduce results. These raw datasets are produced and/or gathered from instruments, sensors, a wide variety and a huge number of distributed programs and tools (are used managing collaborative data analysis, data processing and knowledge discovery), as well as a variety of expensive HPC resources [1, 4, 9, 16].

These raw datasets and data products are shared along with their provenance [16-25] (also referred to as the lineage, pedigree, or audit trail) information. They are shared by publishers and funding agencies [16], in order to reduce the cost of reproduction, verifying results or in situation where it is impossible to run the experiments again (perhaps due to unavailability of resources).

In this regard, a Scientific Workflow Management System runs a scientist's experiments, and provides experimental results and associated provenance information. If scientists want to validate their hypotheses or verify results by running the workflow, A Scientific Workflow Management System (SWfMS) enables scientists to retrace experiments, repeat experiments and improve the accuracy of their own work (reproducibility). To have a reproducible workflow, sufficient provenance information including data origin and history of data derivation should be recorded. Hence, provenance is a fundamental and key component in a SWfMS. We aim to study provenance systems in scientific workflow in this thesis.

Provenance in the literature means origin or derivation. It demonstrates the sequence of creation of data. In scientific workflows, a provenance system is a dynamic mechanism that collects and manages sufficient levels of static and dynamic workflow information in order to enable reproduction, inspection and validation of experimental results. Therefore a provenance collection mechanism is necessary to collect provenance information during a workflow lifecycle [22]. A provenance system greatly increases the value of workflow management systems beyond simple automation. Moreover, it has the potential to make workflow management systems widely acceptable systems among scientists, because workflow systems can capture complex analysis processes at different levels of detail, in addition to automatizing tasks [26].

A number of high data- and computationally-intensive tasks of a scientific workflow system require scheduling and then execution on the resources of a distributed environment. In this case, a workflow management system is required to handle attendant complexities, especially concerning workflow scheduling and execution. Accordingly, a workflow management system consists of a number of components such as workflow engine and workflow scheduling. In the next section, the concept, main components and responsibilities of a Scientific Workflow Management System will be presented.

1.1 Scientific Workflow Management System

A workflow management system manages a flow of work among participants and resources. It also coordinates user and system participants. The workflow management system concerned with the composition and execution of computations in scientific experiments is called a Scientific Workflow Management System (SWfMS) [7, 12, 13].

We explore scientific workflow systems in the form of dataflow models [27], used in many scientific workflow systems such Kepler [28]. The dataflow model can be seen, as a dataflow graph comprising nodes, and arcs that connect nodes. The dataflow graph has an interpreter that encodes the semantics of execution. The semantics of execution is a set of rules governing node interactions, termed the Model of Computation (MoC) [29]. A MoC models the concurrent execution of repetitive tasks and activities at the workflow design-level. It defines how execution semantics are applied on underlying workflow activities and tasks

[30]. An interpreter (or a director in the context of Kepler workflow systems) applies the given rules of the MoC on the workflow system (or dataflow model) to execute it. The Process Network (PN) model is an important variant of dataflow of interest to us, able to express the notion and concept of a scientific workflow system. We develop a Process Network framework in chapter 3, and we implement our workflow case studies in this framework.

The SWfMS in a scientific workflow system is responsible for all of the process of defining, modifying, managing, monitoring and executing scientific workflows using computerized workflow logic representation ordering the execution of scientific tasks [7]. We study the workflow scheduling, execution engine and controller entities of SWfMS in this thesis.

Determining the sequence of execution of interdependent workflow tasks is handled by workflow scheduling. The decision about this sequence is made based on several parameters such as resource constraints (resources may be restricted by their cost or availability), user-defined Quality of Services (QoS) constraints and workflow model constraints [31-34]. We identify two levels of scheduling in workflow systems. Scheduling in computing environment is the first level that is mainly concerned with availability, performance and cost of resources; it also needs to consider the users tasks' deadline and budget constraints. The second level is an innate scheduling that decides when to schedule the execution of actors, and defines the sequence of task execution based on the order of task dependencies in a workflow system. The primary determinants of this level of scheduling are based on semantics of execution or MoC.

We use the term workflow engine to describe entities in SWfMS that is responsible for workflow execution, in either centralized or distributed environments. The workflow engine employs a defined MoC in order to run workflow instances. It initiates workflow model with input data, a direction plan (scheduling policy) through workflow processes, and then activates the execution of process implementation [7, 12].

We propose a notion of *workflow controller* to describe the parts of a workflow system that interact with the environment in which the workflow system is embedded. The workflow controller is not an explicit entity in all workflow systems; however, the concept of

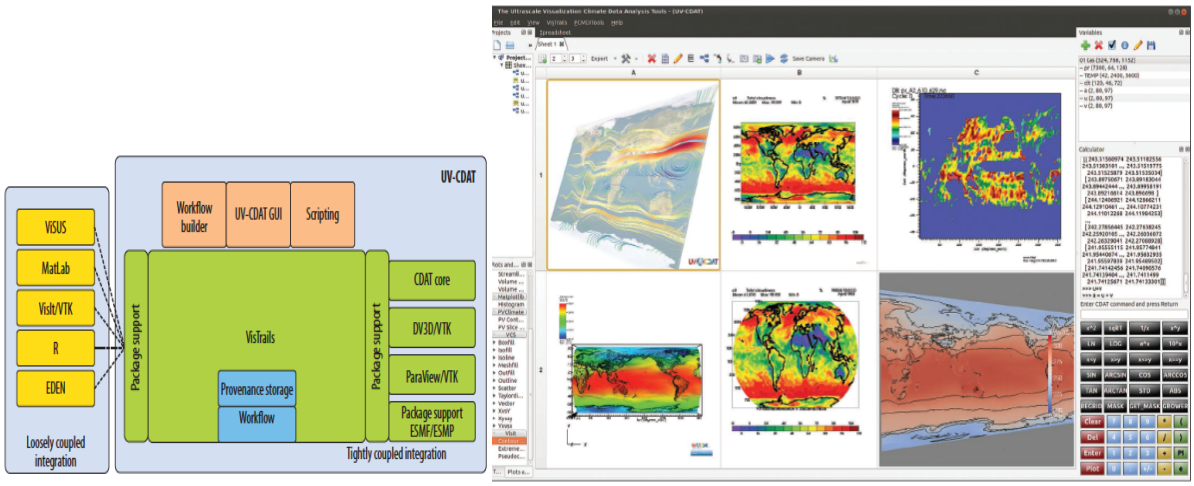
controlling workflow is present in all workflow systems. While some workflow systems manage the running of processes without a workflow controller as a specific entity, there are some aspects that are not directly related to how the workflow runs. These aspects include connection of data input and output with the workflow, and discovering and allocating resources to the workflow. These aspects are more important when a workflow is embedded into a large system. Workflow systems need a mechanism to get environmental information and connect into resources in the environment. Therefore, the notion of workflow controller is about not just the workflow itself, but also the environment in which is embedded. In the next section, a case study of a workflow-oriented and collaborative project is introduced.

1.2 Case study of provenance in a workflow system

In the contemporary eScience [1], there are many varieties of workflow-oriented projects in different areas enhanced with provenance mechanisms such as NEPTUNE [1, 4, 35], Pan-STARRS [1, 4], SenseWeb [1, 36], Earth System Science Server (ES3) (formally known as Earth System Science Workbench (ESSW)) [37, 38] and Ultrascale Visualization Climate Data Analysis Tools (UV-CDAT) [39]. We will illustrate the UV-CDAT in more detail.

As we observed earlier, the information "big bang", or data explosion is a challenge for scientists, who consequently look for a visualization mechanism to effectively understand and use this huge amount of information. Scientists visualize data in order to validate and inspect both expected and unexpected results. Visual data analysis provides facilities to validate theoretical models and to compare it with other models.

Ultrascale Visualization Climate Data Analysis Tools (UV-CDAT) (Figure 1.2.b) is a collaborative project containing more than 70 scientific computing applications and libraries in the field of climate data analysis and visualization. This field faces challenges in terms of big data analytics, reproducibility, heterogeneous data sources, multiple disciplinary domains, incorporation with variety of software and infrastructure, analysis with uncertainty factors [39]. Analysis in the UV-CDAT framework comprises data interpolation, data exploration, parallel processing, hypotheses generation, and provenance capturing [39]. Workflow in the UV-CDAT framework (shown in Figure 1.2.a) is expressed through community tools such as VisTrails, ViSUS, Data Visualization 3D (DV3D), ParaView, VisIt, VisIt, and EDEN.



(a) (b)

Figure 1.2.(a) UV-CDAT framework;(b) GUI for the UV-CDAT based on VisTrails visualization notation [39].

The visualizations and data analyses in the UV-CDAT framework generate tens of terabytes of data. Therefore, a number of efficient approaches (which are defined in ViSUS) is required for exploring and accessing these generated data [1]. ViSUS is a scalable infrastructure for visualization in different applications that is usable from different types of devices. It uses direct streaming and real-time monitoring over large-scale simulation during execution [1]. Systematic documentation is a key requirement for data exploration tools (such as ViSUS), which provides for data preservation, quality determinations, authorship, result reproduction, and result validation. These roles are important when tools have to cope with a huge amount of data and complex analysis processes. The documentation is known as provenance information. In the UV-CDAT framework, provenance information is collected systematically in VisTrails [39-42].

VisTrails [39-42] is a combination of SWfMS and visualization system providing some features such as provenance capturing and management infrastructure; support for computational tasks (visualization and data mining); integration with other applications (such as VisSUS shown in Figure 1.2.b) and libraries, parameter exploration; and results comparison. VisTrails also manages the creation of modules and parameter value changes, plot updating and GUI [39]. Provenance infrastructure is a key feature in VisTrails that captures and maintains historical information about computational steps and data derivation. It captures provenance regarding data products, the workflow that derives these data products and their execution [40]. VisTrails collects provenance in three layers including workflow

instance, workflow execution and workflow evolution. The workflow instance contains the specification of individual workflow while the workflow executions capture runtime provenance information regarding the execution of workflow modules. The workflow evolution records the relationships among different series and versions of created workflows [21]. Provenance in VisTrails enables scientists to reproduce results, solve problems in collaborative way, share and validate results.

The VisTrails *provenance collection mechanism* is implemented by adding monitoring code in ParaView to capture changes in the specification of the underlying workflow [1]. VisTrails is capable of storing collected provenance in MySQL, IBM DB2 and XML file. It also has an intuitive and flexible query interface for provenance queries [41]. VisTrails has a Provenance Browser (shown in Figure 1.3) [43] that enables user browsing through all performed executions. In the next section, we explore provenance collection mechanisms in the context of scientific workflow systems.

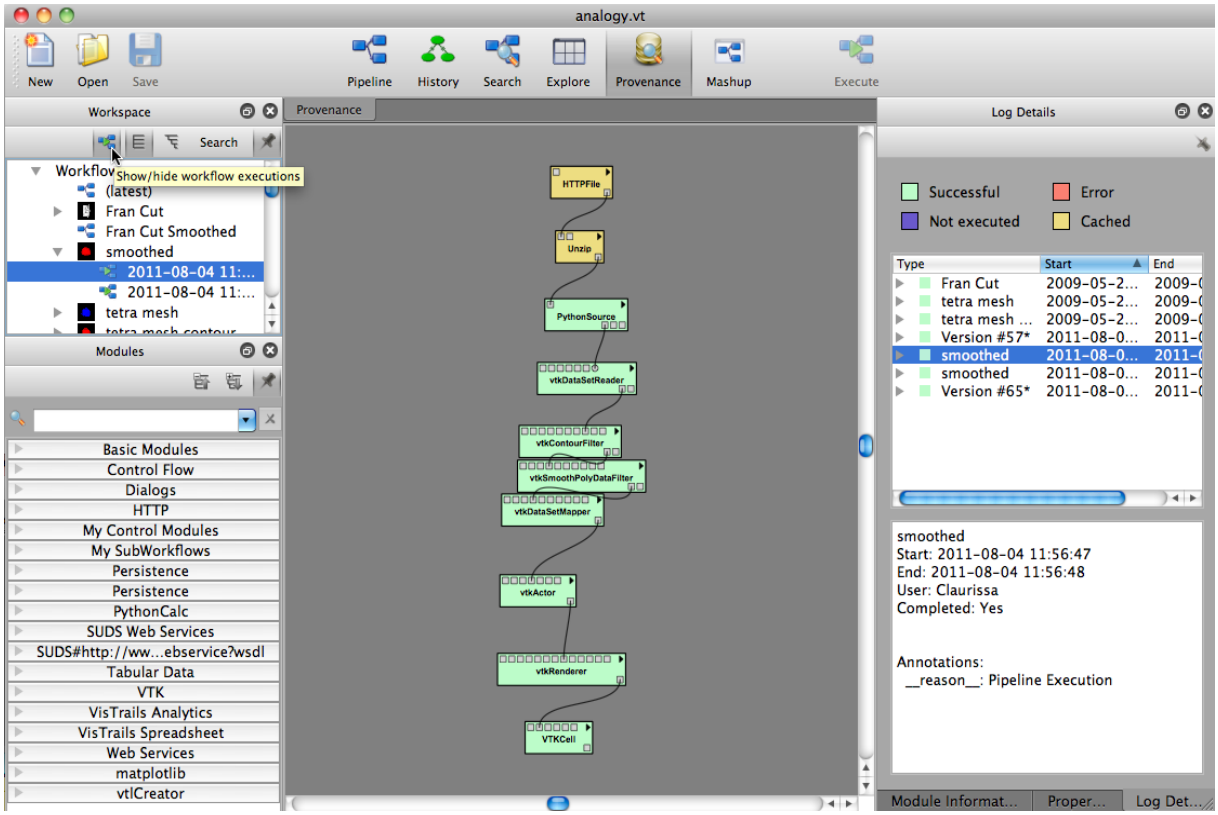


Figure 1.3. VisTrails workflow and (right) Provenance Browser [43].

1.3 Provenance collection mechanisms in workflow systems

Provenance improves the interpretation and understanding of final results by examining the chain of reasoning and sequence of steps that led to final results [21]. In this thesis, we are interested in provenance used to investigate where data comes from and make explicit the sequence of creation of data [21, 44-46] in the context of scientific workflow systems. This is just one of the valid aspects of provenance information. Provenance information is collected through a mechanism of provenance collection. It is a key requirement for provenance collection mechanisms to know what should be collected, and in addition, how collected provenance information should be represented.

Designing and developing provenance collection mechanisms is one of our concerns in this thesis. In the design stage, we elaborate a provenance design description, which determines which sort of provenance information should be collected, at which level of detail it should be collected and how that information is represented. The provenance design description is called the Model of Provenance (MoP), which has a significant influence on the collection mechanism. Therefore, provenance collection mechanisms collect provenance information according to (what is determined as provenance information in) the MoP; and represent them based on (representation format identified in) the MoP. In this thesis, we elaborate a MoP for our provenance collection mechanism capable of representing provenance information at different levels of abstraction. The level of abstraction in provenance information is defined by provenance granularity, including coarse-grained and fine-grained provenance information. We will explore and survey the concept of a MoP in chapter 4.

Although the concept of provenance is clearly a key asset in workflow systems, Scientific Workflow Management Systems and distributed environments, we believe that there has not been a survey of provenance collection in Scientific Workflow Management System since 2009 ((2009)[47], (2008)[21], (2007)[48], (2005) [44, 45]). There have been many novel contributions since that time including major developments in Models of Provenance (MoP) [49-53] and in provenance collection mechanisms for distributed environments [54-58]. Moreover, while provenance collection is reviewed briefly in some previous survey papers [21, 47] (and addressed in [44, 45]), it is our view that provenance collection clearly identified as a first step towards having provenance in workflow systems has not been studied in

sufficient detail to do the field justice. Therefore, it seems that this is a gap that should be addressed, particularly with the respect to variation of provenance collection mechanisms and Models of Provenance in the context of scientific workflow systems. In the following, we will survey provenance collection across a set of scientific workflow projects with an emphasis on provenance collection mechanisms, as presented in chapter 5.

Workflow systems face various changes during execution over the distributed computing environments in which they exist. Therefore, workflow systems can benefit from mechanisms to deal with these changes. The provenance collection mechanism can be seen as an important example of such an adaptive mechanism. In the next section, we explore the ways changes in computing environment can influence provenance collection and the ways provenance collection mechanisms adaptively collect provenance information in the presence of change.

1.3.1 Adaptive provenance collection in workflow system

In this section, we firstly elaborate a workflow system running over distributed environment, and then we explore how provenance collection mechanisms might respond to changes in the environment. Two mechanisms of adaptation that we use in the design and implementation of adaptive provenance collection mechanisms are explained in this section.

A workflow system consists of a number of workflow activities that are capable of running over different environments, and different sets of processing and network resources, as shown in Figure 1.4. Workflow activities could be run over a number of machines, in the local host (presented in workflow activities 3 and 5 in Figure 1.4); remote host in high-performance computing infrastructure such as the Cloud, Cluster and Grid (presented in workflow activities 1, 2 and 4, respectively, in Figure 1.4); or even run in a set of resources that are partly located in local host and partly located in remote hosts. The resources have prices for execution and provenance collection varying based on demand for the resources in computing environments.

Workflow activities in workflow systems are deployed over networks with a variety of characteristics (for example long-distance, high latency, or low bandwidth) using the

infrastructure of distributed computing environments [55, 58]. As shown in Figure 1.4, workflow activities are deployed on several hosts with varying network (or channel) characteristics. The thicker arrows represent better network condition in terms of network characteristics such as latency and bandwidth. For example, the network connection from workflow activity 1 and Cloud environment is in very good condition, possible high bandwidth and low latency, while the connection from Cluster environment and workflow activity 2 is worse.

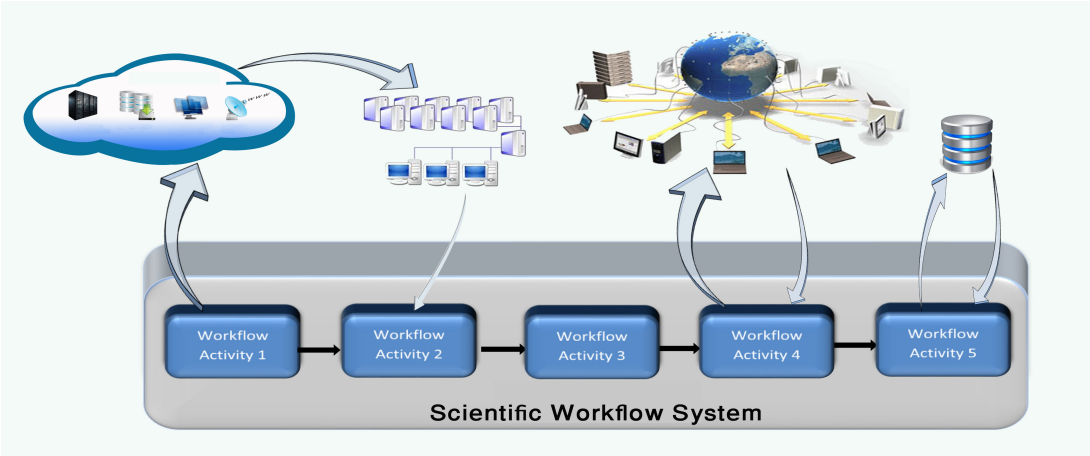


Figure 1.4. A workflow system running over distributed environments.

Workflow systems are submitted with some configuration information including budget and deadline. Workflow systems run over heterogeneous environments with different computing environments, network characteristics, and resources characteristics (such as prices for execution and provenance collection) as explained above. Changes and fluctuations (such as in networks characteristics, resource availability and cost of resources) are in the nature of such heterogeneous environments. These changes and fluctuations have influence on mechanisms operating on the environments including provenance collection mechanisms. The change in the cost of provenance collection of resources (from the usage of computing and storage resources) is an example of possible change in the environment. To see this, if the cost of provenance collection is changed (increased), a provenance collection mechanism can respond by changing the way in which provenance is collected. This might be done, in this case, by using less computing and storage resources to reduce the cost. The provenance collection mechanisms can benefit from adaptively dealing with such changes and fluctuations.

We design and develop adaptive provenance collection mechanisms that adjust the amount of collected provenance information according to changes in environment. For example, the collection of less provenance decreases the usage of computation and storage resources, which results in reducing the cost of provenance collection. This adjustment in the amount of collected provenance is handled through changing the level of provenance granularity. We define levels of provenance granularity through a MoP for provenance collection mechanisms. Our collection mechanism is capable of collecting provenance at different levels of provenance granularity, including coarse-grained and fine-grained (as they are explored in section 4.5).

Coarse-grained provenance is concerned with the sequence of creation of input and output data of workflow processes, while fine-grained provenance is concerned with the sequence of derivation of data within and between workflow processes. The size of data needed for fine-grained provenance is more than the coarse-grained case. In the workflow shown in Figure 1.4, if the channel's bandwidth between workflow activity 5 and its corresponding remote host are decreased, the provenance collection mechanism may collect coarser grained provenance information in order to reduce the amount of provenance information that needs to be transferred to storage in the remote host. If the bandwidth condition improves (turning into a high bandwidth channel), the collection mechanism can collect, represent and transfer finer grained provenance information.

A guiding principle behind how we design and develop adaptive provenance collection mechanisms later in this thesis is *separation of concerns* [59, 60]. The principle of separation of concerns is used to the specification of the handling of core functionality from specification of other behaviours in provenance architectures. Functional concerns are expressed in the base-level application. We implement a Process Network Application (PNA) framework (as shown in chapter 3) for the purpose of addressing functional concerns. The behavioural concerns in our thesis are expressed in two realization ways, including reflection (such as MetaObject Protocol (MOP)) [60-68] and Aspect-Oriented Programming (AOP) [59, 60, 68-70]. There are many intermingled concerns in each application such as workflow systems. We aim to separate and untangle Provenance-collection and Adaptive-granularity concerns that are related to adaptive provenance collection mechanisms. This separation is explored in adaptive architectures for provenance and scientific workflow systems (refer to chapter 6).

We design (in chapter 6) and implement (in chapter 7) two adaptive provenance collection mechanisms with MOP and AOP, capable of collecting provenance information in different levels of provenance information. The format of the MoP is compatible with the generic Open Provenance Model (OPM) (explored in chapter 4) to make the collected provenance information of our adaptive provenance collection mechanism interoperable and usable in other provenance systems.

1.4 Thesis Contribution

This thesis makes the following contributions

- The primary contribution of the thesis is the definition of an architecture for adaptive multiple granularity provenance collection, supported by implementation of a simplified case study (Chapter 4 and 6).
- Design and implementation of an adaptive-granularity provenance architecture based on the principle of separation of functional and behavioural concerns, using two ways of realizing that separation: MetaObject Protocol and Aspect-Oriented programming (Chapter 6 and 7).
- A study of the various Models of Provenance (MoP), leading to a new view of multiple-granularity provenance (Chapter 4).
- Strong OPM compliance in our adaptive multiple-granularity MoP, designed for interoperability, and demonstrated by a case-study implementation (Chapter 7).
- Identifying and defining design dimensions for provenance collection mechanisms, in order to survey existing provenance collection mechanisms in scientific workflow systems (Chapter 5).
- Refinement of an earlier meta-object protocol based Process Network implementation, into a Process Network Application (PNA), representing a Process Network, as a representative scientific workflow system for a case study (Chapter 3).

1.5 Thesis Outline

Chapter 2 contains a literature survey covering provenance, SWfMS, workflow principles in terms of execution and control, and principles of adaptive mechanisms to establish an adaptive workflow architecture. We examine the constituent components of adaptive provenance collection mechanisms in workflow systems.

Chapter 3 describes MoCs in scientific workflow systems for expressing workflows, specifically in the form of Process Networks. The idea of expressing Process Networks as variants of workflow systems adapts from existing applications such as Kepler, which is explored in this chapter. A Process Network Application (PNA) is introduced to represent our implementation of scientific workflow system in this thesis.

Chapter 4 discusses a Model of Provenance (MoP) for expressing what provenance information is collected, and how it is represented and structured. A number of separate MoPs are reviewed and discussed to provide a clear understanding of the MoP concept. We design a MoP representing multiple-granularity of provenance information in a format compatible with the Open Provenance Model (OPM) to make this MoP interoperable.

Chapter 5 puts forward design dimensions that we find useful for categorization of provenance collection mechanisms. A number of provenance collection mechanisms, in the context of workflow systems, are presented and compared based on these design dimensions.

Chapter 6 illustrates an adaptive view of design in workflow architecture for provenance collection mechanisms. It contains descriptions of scenarios that an adaptive provenance collection mechanism is designed for. The adaptive provenance collection mechanisms, designed by MOP and AOP design principles, are explored in this chapter with consideration of provenance and adaptation concepts.

Chapter 7 defines the implementation of the adaptive provenance collection mechanisms in chapter 6. Our MOP oriented adaptive provenance collection mechanism has provenance-collection, adaptive-granularity and distribution meta-behaviours, which are implemented in Enigma. We adapt the Enigma meta-level application to operate our PNA framework. Our AOP oriented adaptive provenance collection mechanism has provenance-collection, adaptive-granularity aspects, which are implemented in the AspectJ framework. Our AOP oriented adaptive provenance collection mechanism operates on the PNA framework.

Chapter 8 summarises and concludes the thesis and provides a discussion of future work.

1.5.1 An “Aspect-oriented” Thesis Outline - the Concerns of the Thesis

In this thesis, we consider several inter-related topics. Their discussion is inevitably intermingled in the chapters of the thesis. Noting that the aspect-oriented view is defined as "allowing the separation of cross-cutting concerns", we think it useful to take such a view of our thesis structure, so that we can look at the different topics, or concerns, of our thesis individually.

Here, we identify the most significant concerns of our thesis, and look at how each of them is developed across the chapters of the thesis. The main concerns addressed in this thesis include, Provenance, Scientific workflow systems, Provenance collection mechanisms and Adaptive provenance collection mechanisms.

1.5.1.1 Provenance

- Chapter 1: The definition and motivation of provenance systems in scientific workflow systems are briefly introduced.
- Chapter 2: We explore definitions and applications of provenance in various eScience domains.
- Chapter 3: the provenance collection mechanism in the Kepler workflow system is explored.
- Chapter 4:
    - A number of Models of Provenance (MoP) are introduced and surveyed.
    - Interoperability of provenance is introduced in the context of Open Provenance Model (OPM). The interoperability of various provenance systems is investigated.
    - A Model of Provenance suitable for our implementation is designed to represent various levels of provenance granularity. It is designed in a format compatible with Open Provenance Model (OPM) for interoperability purpose.
- Chapter 5:
    - Mechanisms of provenance collection, which is one of the main areas of interest in this thesis, are characterized in a number of provenance design dimensions, and these design dimensions are examined in various provenance collection mechanisms of scientific workflow systems.
    - The way reviewed provenance collection mechanisms support Open Provenance Model (OPM) are explored.

- Chapter 6:
    - Provenance as a non-functional concern is defined in both MetaObject Protocol (MOP) and Aspect-Oriented Programming (AOP) software techniques.
    - Provenance contributes in the design of MetaObject Protocol (MOP) oriented provenance collection mechanism as a Provenance-collection Meta-behaviour, and as Provenance-collection Aspect in Aspect-Oriented Programming (AOP) oriented provenance collection mechanism.
- Chapter 7: Our multiple-granularity Model of Provenance and the way provenance is collected in our case-studies is explored.

1.5.1.2 Scientific workflow systems

- Chapter 1: The definition and motivation of scientific workflow system are explored.
- Chapter 2:
    - The basic concept of Scientific Workflow Management System is defined and its reference model is investigated
    - Scientific Workflow Management System in terms of its scheduling, execution and control aspects are investigated.
- Chapter 3:
    - The concept of scientific workflow s expressed an a form of dataflow model, and its semantics of execution is introduced with a simple case study.
    - Process Network is shown to be an adequate representative of scientific workflow system. This view is justified from existing scientific workflow system such as Kepler.
    - A Process Network Application (PNA) is developed as an implementation framework for Process Network.
    - The case studies of scientific workflow used in this thesis are presented.
- Chapter 6: An adaptive workflow architecture is explored.
- Chapter 7: The implemented case study in Process Network Application (as a variant of scientific workflow system) is integrated with a MetaObject Protocol framework and AspectJ framework to develop adaptive provenance collection mechanisms.

1.5.1.3 Provenance collection mechanisms

- Chapter 1: The basic concept of provenance collection mechanism is briefly introduced.

- Chapter 4: A simple mechanism of provenance collection is explored step by step based on a simple Model of Provenance in a simple dataflow graph.
- Chapter 5:
    - Design dimensions of provenance collection mechanisms are defined to provide clear characterization of collection mechanisms.
    - A number of provenance collection mechanisms capable of operating on scientific workflows are surveyed.
    - The explored mechanisms of provenance collection are evaluated and compared based on the design dimensions presented in this chapter.
- Chapter 6: Two provenance collection mechanisms capable of adaptively adjusting level of provenance granularity are designed.
- Chapter 7: The provenance collection mechanisms in chapter 6 are implemented.

1.5.1.4 Adaptive provenance collection mechanisms

- Chapter 1: An adaptive provenance collection mechanism and its motivation are briefly introduced in the context of provenance systems.
- Chapter 2: Two mechanisms of adaptation - MetaObject Protocol (MOP) and Aspect-Oriented Programming (AOP) - are introduced and various aspects of their design and application are studied.
- Chapter 6:
    - Three scenarios are presented as motivations for adaptive provenance collection mechanisms in adaptive provenance architectures.
    - Architecture and desirable design dimensions of an adaptive provenance collection mechanism are presented.
    - Provenance-collection, distribution and adaptive-granularity meta-behaviours of the MetaObject Protocol (MOP) oriented adaptive provenance collection are designed based on design principals of a MetaObject Protocol framework.
    - Provenance-collection and adaptation-granularity aspects are designed in an adaptive provenance collection based on Aspect-Oriented Programming (AOP) design principles.
- Chapter 7:
    - The designed meta-behaviours of MetaObject Protocol (MOP) oriented adaptive provenance collection are implemented in a MetaObject Protocol framework.
    - The designed aspects of Aspect-Oriented Programming (AOP) oriented adaptive provenance collection are implemented.

2 LITERATURE REVIEW

In this thesis, we are concerned with designing and implementing an adaptive provenance architecture applied to scientific workflow systems aiming to adaptively collect provenance information. This adaptation is a response by the provenance system to changes in the environment in which it is operating, such as availability and variation in network bandwidth (see section 2.3 for further explanation). Therefore, this chapter aims to provide the reader with a sufficient understanding of provenance, workflow systems and adaptive concepts.

Section 2.1 reviews the concept and principles of provenance in general terms. We consider two categories of provenance systems, based on the domains and layers in the software architecture to which they are applied. Categorizations based on the criteria of the eScience domains are explained in this chapter, while categorization based on software architectural layers is introduced in this chapter, and explained further in chapter 5.

Section 2.2 explores the concept of scientific workflow management systems. We investigate the requirements of scientific workflows and reference models that are proposed to address these requirements.

In section 2.3 architecture and design principles of workflow system are reviewed. In this section, a conceptual architecture for dynamic aspects of workflow systems is presented. These dynamic aspects of workflow systems are control and execution. The execution in workflow systems is elaborated in the two important concepts of workflow scheduling and workflow engine (explained in section 2.3.1 and 2.3.2 respectively). The notion of workflow controller is an intermediate entity between the workflow system, the user and the workflow execution environment. The concept of workflow controller is used in later chapters in our reflective provenance collection architectures.

Finally, in section 2.4 we outline the required concepts for understanding adaptive architectures in workflows. We explore two mechanisms for adaptation -MetaObject protocol and Aspect-Oriented Programming mechanisms. The design principles and application of these two techniques in detail are explored in this section. Firstly, we explore reflection as the

basic architecture of MetaObject protocol. Principles of reflective architecture and its application to programming languages, middleware and workflow systems are also explained. Following that, principles of Aspect-Oriented Programming are explained and AspectJ as an implementation framework is introduced. This section provides the required background to understand the frameworks for adaptation described in chapters 6 and 7.

In the following chapters, the concept of provenance is investigated in more detail to provide an understanding of the notion of model of provenance in workflow systems. Following that, the design dimensions of provenance collection mechanisms are identifying and compared in various collection mechanisms in the context of workflow systems. Later chapters of this thesis discuss our vision for the design viability and the implementation of adaptive provenance collection mechanisms in scientific workflow systems.

2.1 Provenance

In this section, the concepts and principles of provenance information are presented. Following that, provenance systems are categorized based on the use of those systems in different eScience domains. A number of related provenance systems are elaborated on the categories in this section. To conclude, a classification of provenance systems is presented based on software architectural layers to which they could be applied.

2.1.1 Provenance Concepts

Provenance was originally used in art in order to express the ownerships of a piece of artwork. The provenance of an artwork consists of accurate documentation of its past tracking process (including the chain of ownership) and a history of curators and collectors [71]. Much more recently, scientists realized how digital provenance and related issues could be important in different areas of science [72].

According to the fourth paradigm [1], scientific experiments will reuse generated and shared raw datasets, and also reproduce final results. Therefore, the raw datasets (gathered from instruments and sensors [1, 4, 16]) and results are shared along with their provenance (also

referred to as the lineage, pedigree, or audit trail) information by publishers and funding agencies [16], in order to enable or reduce the cost of reproduction.

Provenance in a scientific computational process preserves the influence of actions on final results, and how tasks were accomplished, which makes provenance as important as the final results themselves. Therefore, the actual published results consist of final results and provenance information. This information regarding exact followed procedures, captured data, annotation, documentation analysis are derived from published results which enables scientists to validate procedures, and to reproduce and extend results [73]. A *scientific computational process* deals with many complex steps and computing resources dealing with data and tasks. The complexity of these computational process is increasing, thus it gets harder and more error prone to keep track of all the computational steps. For a computational process, it is necessary and more efficient that keeping and collecting all the tracks of steps and data involved in a process are handled by the computational process itself [73] facilitated by a sophisticated mechanism of provenance collection. Provenance in the context of scientific workflow systems, as mentioned in the introduction, is defined as a mechanism of collecting, representing and managing sufficient levels of static and dynamic information of workflow systems so as to reproduce, inspect, ascertain fidelity and validating experimental results.

A primary purpose of collected provenance information is to understand what happened in the system during a specific process in the past. The collected provenance can be used to answer questions regarding the runtime decisions, errors, and performance characteristics [24, 71], for example some common queries [24] are as follows.

- Where do the inputs and/or outputs of this run come from?
- What were the inputs, outputs, and/or parameters of this invocation?
- What data and/or invocations were used to derive this output?
- What data was derived from this input?
- Were specific data and invocation dependencies satisfied?
- How was the data created?
- Who was the creator of the data?

In order to further define the range of queries, more queries were proposed in the Third Provenance Challenge [53, 74]. In this thesis, we are interested in answering questions about

the sequence of creation of data, where data comes from or the history of data products in the context of scientific workflow systems. However, there are also other valid questions (some of them are mentioned above) under the general definition of provenance, which are not within the scope of our study.

As we will explore in chapter 4, a provenance system has three aspects [26] including, a collection mechanism, representation mechanism of provenance data model, and infrastructure for storing, accessing and querying provenance [26]. A number of provenance systems are reviewed and summarized in section 4.3 to clarify these aspects. We will focus on the collection and representation aspects in this thesis.

As we will explore in section 2.1.2, provenance is broadly used in many areas and applications [45], while we focus on provenance as used in workflow systems. In the following, some general views on provenance are presented.

Moreau et al. [75] conventionally divided provenance research field into workflow and data provenance. Workflow provenance refers to provenance regarding the workflow system and treats workflow processes as black boxes. This type of workflow provenance is of interest in the eScience and Grid community [75]. It is referred to as coarse-grained provenance, representing the interaction between workflow processes, as a form of derivation between input and output data of workflow processes; or the order of workflow processes' execution. Data provenance is the provenance of interest in the database community. This type of provenance represents detailed interaction and dependencies between data objects. In these models, the sequence of creation of data in system's components is captured at the database level [75]. Data provenance can be referred to fine-grained provenance. Data provenance treats systems (such as workflow systems) regardless of their structure (such as workflow structure), therefore workflow systems abstract from workflow structure.

In the dependency-oriented view of provenance in workflow systems, the workflow components are treated in three ways, as black-, white- or grey-box, during the capture of provenance dependencies [20, 24, 76, 77]. In this view, the provenance is defined as a set of dependencies between data objects. The dependencies between data objects are determined through the logical order of event in event logs. In the dependency-oriented view usually,

provenance collection mechanisms can implicitly capture provenance information in an event log (for example the Taverna workflow system). In the event log, the each event and its corresponding data (either used in read or write events) are captured during a particular step in the workflow run. It captures the event's logical order, and the corresponding data and process for each event. It is possible to look at the order of events (or dependencies between data objects) at three different levels of abstractions as follows:

- *Black-box Model* contains dependencies between workflows at a high level of abstraction and treats a workflow process as a black-box. In this model, the interactions between processes are collected as dependencies between data objects [20, 24, 76].
- *Grey-box Model* contains dependencies between workflows at a high level of abstraction, in addition to some additional annotations on inputs and outputs. It treats workflow processes as black-box while there are some constraints on processes' inputs/outputs, to reveal some process's internal tasks. In this model, the interactions between processes are collected as dependencies between data objects, while it contains additional levels of specification of tasks, in addition to the underlying task's implementation in workflow systems [20].
- *White-box Model* contains dependencies between data objects in database view level of abstraction, which is called as a white-box model. A white-box model treats data objects in the database view level, which is different from the black-box model. It collects the dependencies between database level data objects [20, 24, 76].

Because the terms coarse-grained and fine-grained provenance are used with diverse meanings in the literature, we clarify different interpretation of the terms (especially fine-grained) for use in the rest of the thesis. We consider the hierarchical structure of workflow systems in the definition of coarse-grained and fine-grained provenance in the context of scientific workflow systems. The level of provenance granularity can be coarse-grained that shows the history of interaction between workflow components and the sequence of creation of input and output data of workflow components (see section 4.5.1.1). Fine-grained provenance represents provenance information about the inside of workflow components, in addition to the interaction between components (see section 4.5.1.3). Fine-grained provenance contains additional provenance information about the data produced and consumed; and computational steps inside processes, in addition to coarse-grained provenance information. Coarse-grained and fine-grained provenance are explored in detail in chapter 4.

Provenance collection mechanisms benefit from having a clear description of what provenance information should be collected and how the collected information should be represented. We formulate this description as a Model of Provenance (MoP). The concept of Model of Provenance is elaborated in section 4.1 and 4.2, and in section 4.3, we review and summarize a number of models. For example, a decision about the level of collected provenance granularity is one of the design issues that influence the Model of Provenance, as explained in section 6.3.3 and 6.4.2.

Provenance systems are applied widely on a variety of scientific domains implemented in different layers of software architecture. There are a number of categorizations related to provenance systems presented in the literature [45, 71, 72, 78]. The provenance systems in this chapter are categorized according to their usage in *eScience* and *software architectural layers* (presented in section 2.1.3). In the next section, a classification of provenance systems is presented based on the eScience domains to which they are applied.

2.1.2 Provenance Systems in eScience

Provenance systems have been employed in a variety of domains. Groth [71] divides provenance systems based on the area in which they are used. In this section, each domain and some related provenance systems are explained.

- Version Control Systems
- Application Specific Systems
- Operating System Level Provenance Systems
- Provenance in Database Systems
- Distributed Debugging, Monitoring and Recovery Systems
- Workflow-centric Systems

2.1.2.1 Version Control Systems

Version Control Systems maintain and keep track of multiple versions of documents and files. Concurrent Versions Systems [79] and Subversion [80] are famous systems used broadly by software developers. Version Control System and Versioning file systems can keep user

descriptions of every committed modification of a particular version. The provenance information resulting from those versioning systems is incomplete because they are not able to provide comprehensive information of the generation and modification process of documents and files. They primarily rely on user annotations and description on the new version. Wayback [81], Elephant [82] are some versioning file systems that track changes in this basic way [71].

2.1.2.2 Application Specific Systems

Application Specific Systems are mostly designed for a particular application or domain. The following are a number of projects in this categorization: Geographic information System (GIS) [83], Array Manipulation Language (AML) [84] and Earth System Science Server (ES3) [37, 38].

The Earth System Science Server (ES3) (formally Earth System Science Workbench (ESSW)) is a project that manages Earth science data product that is derived from satellite remote sensing [38, 78]. Global satellite imagery environmental model derives a very large earth science dataset. Scientists in this field apply their algorithm on the dataset to produce a high-level data product. Scientists need a new approach to disperse these datasets among their communities instead of just storing them in a centralized repository. ES3 monitors applications' interactions with their execution environment. During this monitoring process, the interactions (a form of provenance information) such as file I/O, system calls and arguments are stored in the ES3 database. The database is able to trace forward and backward through a provenance graph which is produced based on captured provenance data [21, 37]. The provenance collection process in ES3 is handled by Probulator and the ES3 Core. The Probulator transparently monitors the execution of complex scientific applications. The Probulator uses two applications, namely logger and transmitter. The logger is responsible for monitoring, logging and interacting with the running programs [37].

2.1.2.3 Operating System Level Provenance Systems

Operating System Level Provenance Systems capture the program execution and dependencies between programs at the OS level. Provenance-aware Storage System (PASS) [72] and Transparent Result Caching (TREC) [85] are the most important provenance systems working at the OS level [71]. We explore PASS briefly here and we will explain it in detail in section 5.2.3.

PASS [72] is a storage system that automatically maintains and records the provenance of files by observing system calls. The provenance records are structured in an attribute/value pairs, in which an attribute is a unique identifier, and the value component could be either a simple value or an object reference [72]. PASS accesses to provenance information at OS-level [21]. We explain the mechanism of provenance collection in PASS in section 5.2.3. Muniswamy-Reddy [56, 57] presents three provenance capturing and storing architectures in the context of Cloud with the help of PASS. In these architectures, PASS automatically and transparently collects provenance information, and uses one of the storage architectures to store provenance information.

2.1.2.4 Provenance in Database Systems

Database Systems are very useful for scientists because they often use databases to store and retrieve their experimental data. Therefore, provenance systems working on database systems are very popular, because the provenance systems keep track of provenance information in database system where the data are persistent. In addition, the database systems are very efficient. Database systems enable provenance systems to collect provenance information not only during the execution of the experiment but also after that by scavenging through stored data. In addition, the provenance systems are also able to use the supported querying mechanism in database systems. Trio [72, 86], AutoMed [87] and Panda [72, 88] are some of the provenance systems facilitated by databases [71].

Trio [72, 86] is an observed provenance system that keeps track of used input tuples in order to generate another tuples. It keeps track of the provenance of tuples and supports a data

provenance model. It manages the accuracy and lineage of the data (tuples) as requested by a user or application. Panda [72, 88] is designed aiming to add process provenance for Trio.

2.1.2.5 Distributed Debugging, Monitoring and Recovery Systems

Distributed Debugging, Monitoring and Recovery are a very useful approach to deal with failures, tracing the execution to realize where and why the failure happens and reporting failure and restarting the application after failure. These systems log their events. Provenance can be inferred and produced from the event logs of these systems. DeWiz [89] describes an example of such a system.

NetLogger toolkit [90] and Ganglia [91] are two frameworks that generate events logs on distributed systems. The DeWiz [89] system uses event logs to infer the causal connection between events and generates an event graph model. Performance bottleneck's can be identified by inferring the causal connection between events in event logs. Detailed logs can help a system to find the last stable state of the system in order to execute a rollback-recovery [92].

2.1.2.6 Workflow-centric Systems

Workflow-centric Systems are the most common systems associated with provenance: eight of seventeen provenance systems are used in workflow environments, according to the Provenance Challenge [71, 93, 94]. There is a significant list of workflow systems that are facilitated by provenance systems, such as Kepler [22, 46], Taverna [95], VisTrails [42], Pegasus [18, 96], VIEW [7, 97] and Trident [98] (refer also to section 5.2, in which we consider in more detail, in the context of these works, important design dimensions for provenance collection). Provenance in workflow systems is the main focus of this thesis. A number of workflow systems in terms of their provenance systems and provenance collection mechanism are evaluated in chapter 4, based on design dimensions identified and defined in that chapter. In addition, a reflective provenance architecture for scientific workflow systems is designed and implemented in chapter 6 and 7.

In this section, a categorization of provenance systems applied to different eScience domains is illustrated. In the next section, a classification of provenance systems is presented based on software architectural layers to which they could be applied.

2.1.3 Architectural layers of provenance systems

As mentioned in the previous section, provenance systems have been employed in a variety of domains. Provenance systems have been classified in several categories [45, 72, 78]. Groth [78] identifies four categories which include fine granularity provenance systems, domain specific provenance systems, provenance in database systems, and middleware provenance systems. Simmhan [45] identifies into three categories including *service-oriented*, *database*, and *other data processing architectures* (including command processing and script-based architectures).

Muniswamy-Reddy [72] extends Simmhan's classification [45]. He categorised the *other data processing architectures* category into two sub-categories - *file and file system approaches*; and *source code and build systems*. In addition, Muniswamy-Reddy categorizes provenance collection mechanisms [72, 99] based on automatic collection and application assisted collection. In this category, provenance information is captured transparently to the user and application (without user intervention and application modification). This model of collection is called Observed Provenance, which is used in PASS [56, 57]. The collected provenance information with this model does not have application-specific semantic meaning. Noting that, application specific semantic meaning is captured in application-assisted collection, because the application is aware of what and how it is doing. This model of collection is called Disclosed Provenance, used in Karma, Kepler, and PASOA. The need to modify the user's application is the most obvious disadvantage of this model. Muniswamy-Reddy also puts forward a complementary model of combining the Disclosed and Observed model to automatically and transparently capture the full semantic knowledge of workflow system. This model can be a combination of a workflow engine operating on top of a PASS system, in a way that workflow engine reveals its provenance information to PASS, then all the data stored in a persistent storage [72, 99].

We classify provenance systems based on the particular layer in software architecture where they can be applied [45, 72, 78], such as *Platform*, *Framework* and *Application* layer as described in detail in section 5.1.8.

In this thesis, we are interested in provenance in the context of scientific workflow systems. Thus, it is necessary to introduce concepts and primary literature on the subject of scientific workflow systems. The purpose of information is to build the understanding of scientific workflow system concepts that we will use subsequently in this thesis. In the next section, the concept and requirements of management systems for scientific workflow system and its architectural models are explored.

2.2 Scientific Workflow Management System

Workflow systems need a workflow management system (WfMS) to manage the flow of work among participants and resources; and coordinate user and system participants. This management task is performed based on defined strategies and procedures, which contain a number of tasks. This multiple task management and task coordination consists of a sequence of passing tasks from one component to another component (a sequence of transformations).

The main participants (and resources) of workflows are not usually located in the same logical and physical computer host. A workflow management system needs to integrate all these participants (and resources) together. According to this workflow definition, Hahn *et al.* [13] believe that workflow systems are inherently and conceptually a distributed model [100] and distribution in workflow is a central issue that should not be seen as a coincidental issue. Therefore, in this view, a workflow management system is a fundamental and mandatory component in workflow system context to handle the distribution issue and integration of all participants that may be logically or physically distributed [13]. From the point of view of understanding the interaction between entities in workflows, conceptually it is seen as a distributed system that entities related to each other are in fact distributed.

It is possible to construct a centralized workflow management system (and workflow engine) that can manage and coordinate distributed entities in a workflow. It means that Hahn considers distribution from a modelling point of view rather than an implementation

perspective, so understanding conceptually and modelling the structure of workflow is most naturally done in a distributed manner [13].

Generally, a workflow management system enables specification, execution, reporting, and control on workflows consisting of a number of users, heterogeneous and distributed resources [10]. The workflow management system that is concerned about data and computational composition and execution is called a scientific workflow management system (SWfMS). A SWfMS is a system, responsible for whole the process of defining, modifying, managing, monitoring and executing a scientific workflow. It uses a computerized workflow logic representation for ordering the execution of scientific tasks [7].

The design of a SWfMS at an appropriate level of abstraction poses questions regarding: what are the actual and crucial requirements of SWfMS? and which are the main challenges in designing an architecture for SWfMS? It seems desirable to have a widely acceptable reference architecture, which is proposed from an independent source.

In the following, we discuss a possible reference model for SWfMS. Although, we do not specifically use this reference design subsequently, it has been useful in identifying criteria for design and requirements of a SWfMS. We start this discussion by looking at the well-known Workflow Management Coalition (WfMC) model, originally developed in the context of business workflows, and examine its characteristics, and then we move on to refine further characteristics appropriate to scientific workflow systems.

2.2.1 Reference Model for SWfMS

The Workflow Management Coalition (WfMC), which is an industry-wide consortium, defines a reference model [101], in order to provide a standard architecture for workflow systems. The WfMC reference model is not a suitable model for a distributed environment owing to being a monolithic model [8]. Therefore, a number of works explored this issue including the CORBA architecture [10], the Newcastle-Nortel workflow system [8], and a framework for an adaptive distributed workflow system by Purvis *et al.* [102].

Despite the above-mentioned external architectures that make WfMC usable in distributed environments, WfMC is not an appropriate reference model for SWfMS, because firstly the business and scientific workflow system have different goals and purposes, explored in section 2.3.3. Secondly the business workflow is control-oriented, while scientific workflow is data-oriented which introduces a new set of requirements and challenges.

In the literature [7, 12, 97], requirements have been suggested that scientific workflow systems should satisfy. These requirements include reproducibility, service integrability, data integrability, usability, GUI and user interaction, monitoring, robustness (flexibility), interoperability, and scalability [7, 12, 97]. Among various SWfMSs, which have been developed, a number of them partially satisfy these requirements. However, each of them follows their own workflow languages that are not yet standardized and standardized. They are not developed on explicit architecture design or they build upon other architectures. Finally, they do not use standard or compatible provenance models in either capturing, storing, querying or representing models [7].

It is possible to consider the possibility of SWfMS architecture extending the WfMC reference model to add some extensions (including execution engine, service bus, security components and also some deployment components such as monitoring, auditing and provenance) that are needed to meet a number of scientific workflow's requirements [12]. Therefore, WfMC's reference model does not satisfy the first five requirements [7, 97] of scientific workflow systems. As a consequence the WfMC reference model is not complete and compatible enough to meet all the scientific workflow requirements [7, 97].

A reference model or architecture is one of the crucial and inevitable requirements for designing a SWfMS. This reference model provides guidance for sophisticated SWfMS design in different scientific disciplines and makes it possible to compare different scientific workflow systems. There is no widely accepted reference model for SWfMS that is designed or approved by OMG or any Workflow Communities. Lin *et al.* [7, 97] proposed a candidate reference architecture for SWfMS aiming to address the above-mentioned requirements.

SWfMS will run scientist's experiments and show the results of various workflow at a variety of levels of abstractions for users. The highest level of results should be presented to

scientists. Scientists want to validate their hypotheses by running the workflow. This validation might be in a few steps by a trial-and-error method; therefore the workflow usually would be constructed incrementally to help scientists to prove their processes and eventually their results. Scientific workflows should also be robust and durable to be able to continue their processes even with failures while running (fault tolerance and fault handling). Scientific workflow's results and outputs must be reusable, modifiable, and shareable between different scientists and groups. Apart from the highest level of results presented to scientists, some level of produced and consumed data (intermediate data) and the way they are produced and consumed are captured in the provenance system.

In this section, we have provided an overview of SWfMS and its requirements and reference architecture. In the following section, we present a conceptual architecture for dynamic aspects of workflow system that are control and execution of workflow system.

2.3 Principles of Workflow: Execution and Control

In this section, some general principles of workflow systems are presented to provide a comprehensive overview of SWfMS. In this regard, a classification of workflow is presented and a conceptual architecture is proposed for workflow systems, including workflow engine and workflow controller that are explained in detail in this section. In harmony with workflow scheduling issues, Workflow Engine as a SWfMS component is nominated for the responsibility of execution. At the end of this section, a conceptual view of a workflow controller is presented. We adopt the concept of a Workflow Controller as an intermediate entity between the workflow system, user and workflow execution environment.

Recall in the introduction chapter, workflow systems can be divided into in two broad areas of scientific or dataflow (concerned with data) and business or control-flow (concerned with control) workflows. These two workflow views are substantially different. The way of activation and communication of activities is the primary of difference in scientific or dataflow (concerned with data) and business or control-flow (concerned with control) workflows.

A scientific workflow is explained as a problem-solving environment designed to work with data-driven applications; and it considers the flow of data between/inside workflow activities. This dataflow determines the connection and interactions between activities in workflow from data producers to data consumers [103-105].

A business (or control-flow) workflow is a standard model to primarily define and employ business rules, policies and high-level management in business workflows. Control-flow arranges the order of executed services in workflow systems. It represents the control dependencies and partial ordering relation of business workflow activities. A control-flow is designed to control connection between activities and some control-structures. This can represent a transfer of control from a process to another process [103-105]. In the rest of this section, we present an architecture for scientific workflow systems.

A workflow can be represented as a graph, consisting of nodes and arcs that express relationships between data and operations (transformations) on that data. Workflows present a set of relationships between data and processing, which they express either as a control-flow or dataflow dependency. Figure 2.1 shows a simple workflow system architecture, nodes (producer and consumer) are connected via an arc (channel), which is a buffer to transfer data and control tokens. Produced data from a task in a node is consumed by subsequent tasks through a predefined graph topology. In the context of scientific workflow systems, nodes express the processing of data, and arcs express the flow of data. Some semantics need to be applied to this graph in order to define the execution of this graph. These semantics, called Model of Computation (MoC), can be defined in an interpreter and will be discussed later in chapter 3.

In the workflow system architecture presented in Figure 2.1, the "*Information Service*" component enables users to submit their requests, orders, and customizations through the SWfMS. The SWfMS also communicates with the Information Service in order to collect user requested tasks and environmental information. For instance, the Information Service may provide information regarding the host to store collected provenance information, the level of granularity user requested, and what is the condition of underlying infrastructure resources and network information in terms of bandwidth and network latency.

The SWfMS, shown in Figure 2.1, also contains the Workflow Engine and the Workflow Controller. The workflow Engine runs workflow instances and applies the user-defined model of execution, firing rules and policies in workflow systems, which are most concerned about scheduling strategy. The scheduling that is explored in section 2.3.1, determines the sequence of tasks executed in the workflow system. The Workflow Controller is a part of the management system that needs to communicate with the Information system and workflow instances. It interacts with the environment in which the workflow system is embedded.

The conceptual architecture proposed in Figure 2.1 does not explicitly show provenance. In section 3.2.2 and section 6.2.2, we will discuss the incorporation of a provenance (recorder) in a workflow architecture such as that presented in Figure 2.1.

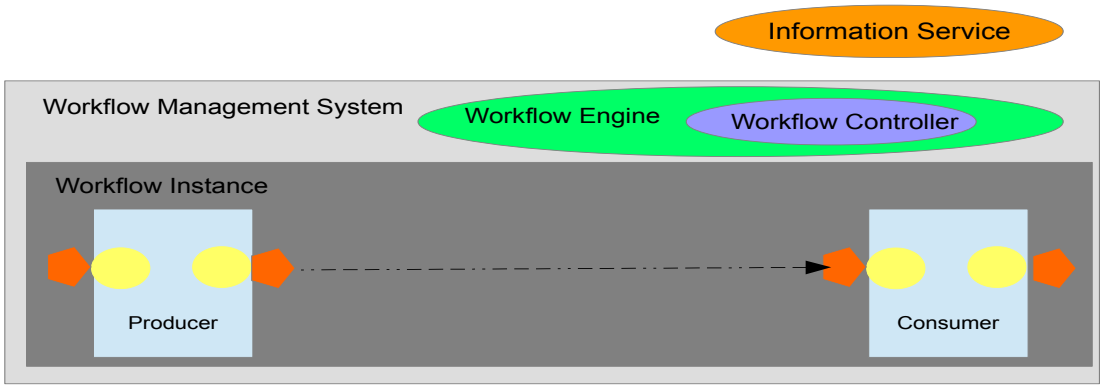


Figure 2.1. Workflow system architecture.

Workflows are inherently distributed [13, 100] (as it is discussed in section 2.2), therefore coordinating the interaction and execution between distributed workflow components should be addressed in an appropriate manner. Orchestration [100] and choreography [100] are two approaches for coordinating the interaction and execution of workflow components. In the rest of this section, we explore the concept of orchestration and choreography, then we explain the main components of workflow architecture including, workflow scheduling, Engine and Controller.

In the orchestration approach, a single controlling entity (client) is conducting all the workflow execution processes with the help of operating message interactions (request and respond pattern) with each of services; the client requests service invocations by sending a message, then response messages will eventually arrive back at the client [100].

Choreography enables collaboration between services to achieve a common goal. Service orchestration has a central control of information and process flow between services (central controller), while service choreography just tracks sequences of messages between multiple partners (decentralize and no central controller). In service choreography, all participating services are treated equally, in a peer-to-peer fashion and services involved in workflow interaction directly communicate with each other. Therefore, when the output data of a service need to be sent to another service, the output data is sent directly without any intervention of and transformation to a central entity.

The choice of orchestration or choreography influence the way the workflow behaves and how data is handled, because it influences the services (actors) between which data flows. Orchestration and choreography influence control in sense that in the same execution in orchestration and choreography, different part of workflow systems are executing and interacting together. However, the way the logical control definition is defined in scientific dataflow does not necessarily change between these two approaches. Therefore, the way they are interpreted and executed by runtime system can be quite different, but the conceptual view that the way workflow is programmed can be same. In the next section, the concept of scheduling is elaborated in the context of workflow systems.

2.3.1 Workflow scheduling

Workflow scheduling is a process of determining the sequence of interdependent workflow tasks' execution based on resource constraints (resources may be restricted by their cost or availability), user-defined Quality of Services (QoS) and workflow model constraints [31-33]. Resources including storage resources, computing resources, and devices are managed in different (centralized or distributed) computing environments such as Grid and Cloud. Scientific workflow systems are generally data and/or computationally intensive; therefore they need to be allocated into powerful resources mostly in Grid and Cloud environments. These environments are naturally dynamic and their resources are autonomous, which may lead to having variation in performance or it may lead to fluctuation in the availability of resources at different times and situations.

User-defined QoS is another criterion effecting the workflow execution. Users in the Grid and Cloud can define an appropriate budget and deadline for their jobs. Last but not least important criteria is workflow model constraints, such as temporal and causality [33]. The workflow model constraints are defining the existence and order of dependencies between tasks in workflow systems, which are not already definable in workflow structure [32, 33]. The temporal and causality constraints are defined in one of the following forms [33], 1) a task must execute; 2) must not execute; 3) if task 1 executes, task 2 must execute as well; 4) if both task 1 and task 2 execute, task 1 must execute before task 2 [33].

Yu *et al.* [31], divided workflow scheduling into best-effort and QoS constraint-based scheduling. The best-effort scheduling mainly focuses on execution time minimization while meeting user's QoS requirements. On the other hand, QoS constraint-based scheduling tries to maximize performance according to defined QoS criteria and meet other, secondary, QoS requirements, such as minimizing the cost under deadline constraints. It is very hard to schedule the entire workflow's tasks in advance or decide about scheduling of entire of workflow tasks, because resources in Grid and Cloud environments do not show the same performance all the time; and also the order of dependencies between tasks in workflows are determined. Therefore, two scheduling approaches, namely task partitioning [31, 106] and iterative re-computing [31] are proposed to efficiently allocate resources in dynamic environments. Deelman *et al.* [106] partitioned workflow tasks into a number of sub-workflows, then resources are allocated into these sub-workflows. This task-partitioning algorithm helps the scheduler to map remaining sub-workflow to resources with updated information just after the execution of the previous sub-workflow that is already mapped. The updated information comes from an Information Service, which is a vital part of workflow systems architecture (shown in Figure 2.1), and also the computational environments architecture such as the Grid environment. Scheduling based on recently updated information makes the scheduling optimized and dynamically adapted with new changes. Iterative re-computing [31] reschedules the unexecuted tasks in workflow whenever environment has changed. In this case, the initial scheduling is used before any changes happen to the environment.

Based on the above-mentioned paragraph, scheduling in workflow management systems is needed to consider both environmental (resources and user-defined QoS) and workflow

(temporal and causality) requirements. Two different levels of scheduling algorithms in computational environment and workflow systems may be required to be designed to meet those requirements. The first level of a scheduling algorithm is in the computational environment, which is mainly concerned with the availability, performance and cost of resources; it also needs to consider the users tasks deadline and budget constraints. The following projects use the computational environment level of scheduling in Grid workflow management system [107] : Condor DAGMan, Askalon, GrADS, ICENI, APST, and Pegasus. The execution of workflow has to be run on physical resources (in computing environment) that are shared with other, unrelated, computing tasks. Consequently, co-opting this process into a workflow imposes the scheduling system of the facility, which is a part of high performance computing environment.

Another level of scheduling is at the level of workflow system that decides when to schedule the execution of actor and defines the sequence of task executions based on the order of task dependencies in a workflow system. Workflow systems inherently build on the semantics of execution (refer to section 3.1 and 3.2, named Model of Computation (MoC)), which are a set of rules governing node interactions. The rules (in the model of computation) and semantics of execution help workflow scheduling to determine how tasks can be executed; however, there are other aspects of scheduling related to how workflow systems work in the computing environment, as explained earlier.

One of the primary determinants of scheduling of task comes from MoC that models the concurrent execution of repetitive tasks and activities at the workflow design-level. MoC explains the underlying operation of the workflow and also defines how execution semantics are applied to a workflow activities and tasks [30]. A director [29, 108] (in the context of Kepler workflow systems) or an interpreter (in general terms) can be regarded as a way of executing of a workflow according to the given MoC. In this thesis section 3.1.1 present a simple experiment that explains the use of a simple interpreter. The Kepler workflow system has an innate scheduling mechanism (employing at the workflow level of scheduling) that is explained in section 3.2.1. In the following sections, SWfMS components, particularly the workflow engine and workflow controller that are shown in Figure 2.1 will be described.

2.3.2 Workflow Engine

The workflow engine (shown in Figure 2.1) represents entities in workflow systems that are responsible for workflow execution. The workflow engine can be operated in either centralized or distributed environments. The workflow engine runs each workflow instance and employs defined model of computation and execution, firing rules and policies in workflow instances. It initiates workflow model with input data, (schedules or) plans a direction through graph processes, and then activates the execution of process implementations [7, 12]. It also as an execution platform able to manage cross-cutting concerns [1] and tasks, such as fault handling in the form of monitoring and recovering; maintaining workflow instance's data and state; invoking service applications, dealing with adaptation of workflow model and instances at runtime; dealing with heterogeneity of computing platforms and data types; managing control-flow and dataflow; handling optimization of data storages and in the presence of concurrent and parallel execution; handling security; and the monitoring and collecting and tracking of provenance. The workflow engine determines scheduling policies to perform execution appropriately.

The workflow engine is the main component of SWfMSs that is responsible for the execution of workflow [7]. The workflow engine provides isolation between the workflow model and workflow run (execution) model, in order to cater for a variation of execution model. This variation depends on different workflow's components, executional resources, and SWfMS's architectures. In the next section, we explore the concept of workflow controller.

2.3.3 Workflow Controller

The workflow controller in SWfMS enables interoperability with heterogeneous environments, and handle environmental changes during runtime. There are components in a workflow system that realizes the workflow controller's function as, shown in Figure 2.1. In this thesis we have a particular focus on the workflow controller because it is an engine for adaptation. We are intending to adopt the notion and concept of workflow controller in workflow systems to design an adaptable workflow architecture that builds above the concept

of workflow controller (presented in chapter 6) to control and manage workflow interactions with distributed environments.

The idea of workflow controller is an abstract notion (notional view) that we propose as a place-holder for a part of a workflow system that looks after how the processes and tasks of workflows are coordinated and executed in the environment in which workflow is embedded. Not all workflow systems have an explicit controller; however, all of them have mechanisms that carry out the function of controlling of workflow. Therefore, the controller may not be a concrete and separate part of a workflow system. This is a primary distinction in our usage of the terms controller and engine: the former is a notional view with its described functionality in different parts of an actual implementation, while the engine is typically a discrete physical realization.

The workflow controller is an interface between computing environment that the workflow is embedded in, and the Information Service informs and updates the workflow controller frequently with the changes in the environment and user requirements. The workflow controller provides a mechanism to get environmental information and also connects workflows into resources in the environment. From this point of view the workflow controller is not just about workflow itself but also the environment in which is embedded. For example, in the case that input (data) need to be prepared from external resources. The workflow controller would manage when and in which order these inputs will be available.

Scientific workflow systems manage and control workflow tasks and sequences. These workflows would possibly deploy over networks with a variety of characteristics (for example long-distance, or low bandwidth); and even using in the Cloud or Grid infrastructure [55, 58]. These workflows may be required to run over different environments and different sets of processes and network resources. Therefore workflow system implementations need to be run over such heterogeneous environments. The workflow system needs an interoperation mechanism between these environments. The mechanism needs to deal with changes that may occur in underlying infrastructure and management of the system during runtime. The workflow controller is considered a part of SWfMS (shown in Figure 2.1) that is responsible for dealing with the changes in the environment, and in addition makes the workflows and its components adapt with the changes.

The workflow controller is an appropriate place to employ an adaptation mechanism into workflow instances according to network information that is collected from the Information Service (shown in Figure 2.1). It also can collaborate with workflow engine to manage extra functionalities and non-functional requirements for the workflow system such as, adaptation, reuse, fault-tolerance, reconfiguration, monitoring, and provenance.

In this section, we presented a conceptual architecture for workflow systems to describe principles of workflow systems. In this architecture, a scheduling, executing and controlling for workflow system are presented. In the next section, adaptive design techniques are introduced for workflow architectures.

2.4 Towards Adaptive Provenance in Scientific Workflow

Later in this thesis, we show how adaptive principles are used to design an adaptive provenance architecture for distributed workflow systems aiming to adjust the granularity of provenance information. In this section to provide a foundation for that, we explain fundamental concepts of reflection and Aspect-Oriented Programming approaches. We also review the application of the approaches in the literature. In the following, we intend to show the relevance of an adaptive design approach to workflow architecture, and in particular its application for provenance in scientific workflows. We want to show that it is productive to apply adaptive design principles to workflow architectures. An adaptive design is capable of reacting to changes in the environment, user requirements or underlying system requirements, as explained in sections 1.3.

An adaptive system aims to adapt the behaviour of an application in response to environmental changes; ideally, it exhibits fast and frequent reaction to the changes in environment [68]. Reconfiguration is one possible form of response to changes in various aspects of a system including structural changes, geographical changes, interface modification and implementation modification. Reconfiguration takes place during maintaining or re-arranging the elements of a system from different parts of it including application, platform, underlying infrastructure and architecture [68]. In this thesis, we do not consider reconfiguration in detail, but focus on other aspects of adaptation.

Parameter adaptation and compositional adaptation are two types of adaptation [60]. Parameter adaptation [60] determines the behaviour of program (or system) through adjusting the program variables or redirecting program to use a pre-existing strategy (adopting a new strategy is not permitted). While, compositional adaptation [60] allows the adoption of algorithmic or structural system components to address unforeseen situations during system development and also adjusts the system to fit to upcoming changes. Compositional adaptation, which is the subject of this thesis, is more powerful and flexible than parameter adaptation [60].

There are three key design techniques that support the compositional type of adaptation. These key design techniques include separation of concerns, computational reflection, and component-based design. The existence of these design techniques provide scope for adaptation [60]. These design techniques are explored in this section. We will address these three design techniques for the compositional adaptation in our work.

McKinley *et al.* [60] claim that indirection of interactions between entities of programs (or systems) is the logic behind all compositional adaptation. There are several software techniques [60, 68] that are capable of supporting a form of indirection of interactions between entities of program. These include software design patterns (such as Wrapper, Function Pointer, Proxies, Strategy Pattern, Virtual Component Pattern) [60, 68], Aspect-Oriented Programming [59, 60, 68-70] and reflection (such as MetaObject Protocol)[60-68]. In this thesis we focus on reflection (refer to section 6.3 and Aspect-Oriented Programming (refer to section 6.4) software techniques aiming to provide a form of adaptation. These software techniques are capable of expressing separation of concerns and also they have originated on the basis of computational reflection. Therefore, they address these two key design techniques of adaptation. In the next section reflection software techniques, its principle and architecture are explained.

2.4.1 Reflective Architecture

Reflective architecture is introduced in a system to provide a mechanism for extra functionality. Reflective systems aren't used directly in solving domain problems, which are

the core concern of systems. In this regard, meta-level programming helps separate the functional and non-functional behaviours of system. This programmatic partitioning of behaviours is called *separation of concerns* [59, 109, 110]. A concern is an area of interest [61] either aiming to solve domain problems directly (business logic of program) or indirectly (extra functionalities). Concerns convey different meanings in different contexts. In a program (especially in OOP language), there are several concerns, such as security, load-balancing and logging, that are tangled. Separation of concerns is a technique to untangle concerns and express them through separated interfaces. As mentioned previously, there are several techniques [59] to employ separation of concerns such as reflection and Aspect-oriented programming (refer to section 2.4.2) techniques that we explore in this thesis. The subject of this section, Reflection, separates the functional (business logic of a system operated at the base-level) interfaces from non-functional (meta-behaviours operated at the meta-level) interfaces of the system.

From the programmer's point of view, reflective systems are implemented using base-level and meta-level interfaces. Application programmers use the base-level interface to implement the functional concern of their application and use the meta-level interface to implement the non-functional concerns. The means of communication between base-level and meta-level objects are determined by a MetaObject Protocol. We will use these concepts and mechanisms (such as MetaObject Protocol) to design and implement our reflective provenance collection architecture in scientific workflow systems.

2.4.1.1 Principal concepts of Reflective Architecture

A system (or program) comprises of a number of separate modules that handle the main functionality of the system aiming to contribute directly in solving problems in a domain. The modules that are the constituent objects of the system are different in different systems according to their purpose; and the design and architecture of each system. The modules in object-oriented system are objects and, in component-based systems, modules are components.

It is often desirable to extend the defined system to a computational system [62] (also simply called system subsequently) that not only solves the problem as before, but also provides a

mechanism for extra functionalities for the system such as answering questions about and/or supporting actions on some domains that it works on. Therefore, computational systems are facilitated by computational objects (that performs computation about external domain problem) and reflective objects [62]. Reflective objects (reflective computation) perform computations about themselves and they do not directly contribute to the solution of problems [62]. They aim to help computational objects improve and extend their functionalities. Reflection is a technique that allows a system to maintain information about itself and use this information to change its behaviour [63].

A reflective architecture can be defined for each computational system. It separates the external domain functionalities of system (functional concern of system) from its external functionalities of system provided by reflective objects (non-functional concern of system). In the reflective architecture, each computational object (which, in the following, we will call object) in a computational system (which, in the following, we will call system) is associated with one or several reflective objects (which, in the following, we will call meta-object).

A meta-object is a structure that holds all the required reflective information for the object (computational object). In a reflective language context, meta-objects contain information about the implementation and interpretation of objects to show how object handles non-functional concerns such as methods for logging and creating new instances [62]. In component-based context, the meta-object contains information about the design and implementation of components such as, methods to specify the name of component, properties, list of methods, how to access and/or modify methods, how to access and/or modify the structure of components, and how to observe and/or modify the way components are connected together. Therefore, meta-objects contain a number of concerns (behaviours) that present either the structural or behavioural concerns of the system. These are called non-functional concerns or meta-behaviours (defined in section 2.4.1.2).

A system developer uses reflective design by implementing two interfaces to identify the difference between, what a system does (the base level interface) and how it does it (the meta-level interface) [111]. Application programmers use the base-level interface to implement the functional concern of their application, while meta-level programmers use the meta-level interface [64, 111] to implement the non-functional concerns (such as persistence, security,

naming and communication infrastructure [64]) of that application. Separation of concern is one of the primary design principles of reflective systems obtained by partitioning the system into application layer (functional and base-level) and non-functional layer (meta-level) [64, 111].

Reflective system designed by a reflective architecture separates the functional and non-functional behaviours of a system, which explain what the program does and how it does it. The separation of the functional and non-functional behaviours in reflection as is defined in a MetaObject Protocol (MOP) technique allows programmers to focus on the specific behaviour of system and customize it [111]. Reflective systems are useful in wide range of purposes. For instance, they can provide,

- Monitoring mechanism in the form of documentation, log, history and explanation of large systems for users that are dealing with the complexity of large systems [62].
- Observation and modification mechanisms to inspect the status and structure of the underlying object environment [64, 66, 67, 112, 113].
- Reasoning about control mechanism to preserve the status and behaviour of data items and activates processes when specific events occur [62].
- Reforming mechanisms to add or remove some properties to the objects either in the structural or behavioural aspects of the object's environment [64, 66, 67, 112, 113].
- Tracking mechanisms to track relations among objects, such as dependencies, constraints and provenance [62].
- Self-optimization, self-modification, and self-activation [62].

The MetaObject Protocol (MOP) is introduced in the next section. A MOP is used to design and implement (presented in chapter 6 and 7 respectively) an adaptive architecture for provenance collection in workflow systems. In the next section, concepts and design principles of reflection are expressed in terms of a MetaObject Protocol.

2.4.1.2 MetaObject Protocol

A MetaObject Protocol (MOP) allows programmers to focus on the specific behaviour of the system and customize it [111]. We refer to the specific behaviour that MOP is focused on as a meta-behaviour. The MOP determines two mechanisms of reification and reflection for the

collaboration and communication purposes of base-level and meta-level objects, as explained below. Meta-behaviours are introduced to the system through reification and reflection mechanisms in MOP. In this section the principle concepts and mechanisms of MOP are elaborated.

Smith [65] identifies the following requirements for a system to be considered as a fully reflective system,

- *Self-representation* is the representation of itself that should be simple and complete. Self-representation is the first requirement of a reflective system which should exist to provide some information in meta-level (reflective information) for the system.
- *Causal connection* between self-representation and the system. This requirement causes, if any changes happen to the self-representation, these changes should also effect the system and vice versa.
- An appropriate *vantage point* to safely stand and perform reflective works. It seems necessary to modify running program and execute the reflection code in a safe time and place to avoid damages that may arise from incomplete changes [61, 65].

The defined requirements above are the basic design issues that should be considered in understanding and designing a MOP. According to the requirements defined by Smith for fully reflective systems, it seems necessary to have a mechanism for causal connection in most reflective systems, which concerns with modifying base-level objects according to changes in corresponding object in meta-level. These reflective systems are interested in modifying, adapting, reconfiguring, and customizing (base-level) systems according to the changes happening in the environment and/or requirements of the system; in order to make the systems more flexible, reliable and efficient. In this regard, objects and meta-objects, located in the base-level and meta-level, need to collaborate in appropriate ways based on a defined protocol for their communication in the MOP. The MOP exposes selected aspects of a system that programmer is interested in. Therefore, the selected aspects are accessible and modifiable through MOP mechanisms.

As mentioned earlier in this section, reification and reflection mechanisms in MOP introduce meta-behaviours to the system. The communication and relation between base-level objects and meta-level objects concerning the meta-behaviours in the reflective architecture is determined in MOP through two mechanisms of reflection and reification, as it is shown in

Figure 2.2. The reification mechanism identifies how the structure and behaviour of base-level objects are transferred to meta-level objects while reflection mechanism identifies how the meta-level objects observe and/or customize the base-level objects. In the reflection design, each base-level object has one or a number of associated meta-objects in a sense that each meta-object is independently responsible for a specific aspect (or meta-behaviour) of system [112, 113].

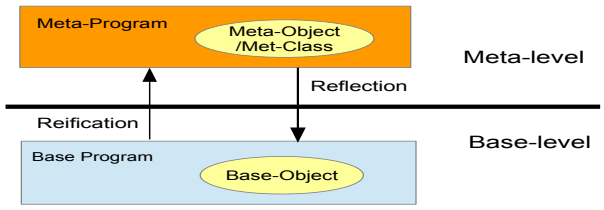


Figure 2.2. The relationship between reflection and reflection.

Reflection is the ability of a computer program to introspect and intercept the structure and behaviours (e.g., the values, meta-data, properties and functions interception) of an object (or several objects) at runtime [67]. The introspection mechanism could observe and query the current state of the system, while the interception mechanism makes changes in the system behaviours at runtime to satisfy the current operating environment [66, 67]. The system behaviours are presented by ways of the system constituent components are communicating, which can be in a form of method invocation or message passing. There are three commonly used interception (intercession) mechanisms on invoked methods. The intercession strategy identifies points at which the message is exposed to customization.

- The *before-after strategy* performs the relevant action immediately before or after the method invocation. It is used in the AspectJ (explained in section 2.4.2.2), OpenCOM and OpenORB (explained in section 2.4.1.5).
- The *trap strategy* provides special trap methods in the target meta-object.
- The *decomposition strategy* breaks the message or method invocation into smaller traps. It will be explained and used in section 6.3.2.1.

Reification is the mechanism of concretizing an abstract idea regarding computer program into an explicit and tangible data model. Reification makes it possible to explicitly formulate concepts and ideas, which were implicit and not easily expressed in computer systems. In other words, reification is a process of making a behaviour into a first-class citizen. A first-class citizen (e.g. object, entity, or value), in the context of a particular programming

language, is an entity that can be constructed at runtime, passed as a parameter, returned from a subroutine, or assigned to a variable [114]. The reification of a computational model in base-level of a system consists of structural and behavioural model of reification [111] (in some literature such as [115] referred to as reflection).

Structural reification is the process of converting some element of a system (such as an object, class or method) into a meta-object [111]. It enables systems to reify the current executing program, in terms of language methods and states. In this type of reification, a programmer can intercept and inspect the functionality of the program and model the domain [67, 116]. Structural reification reifies the base-level objects (known also as base-objects or objects) to meta-level objects (which could be in a form of meta-objects or meta-classes as shown in Figure 2.3 and 2.4). A meta-class is a type of meta-level object that has structural descriptions about base-objects. The objects of a meta-class (called meta-instance) show the same behaviours [67, 115].

In the meta-object model, the reflective information about implementation and interpretation of these base-objects (that is structural description of base-objects) resides in meta-level objects (either meta-objects or meta-classes). In a situation that meta-level objects are modified as the base-level objects are changed, the systems are called *causally connected* [67, 115]. In the context of object-oriented programming (OOP), base-objects (including class, method, inheritance relationship and/or other aspects of object structure) are reified structurally to meta-level objects (called meta-instances) holding the reflective information about implementation and interpretation of the base-objects. In conventional programming, this information is implicit for meta-level programmer while meta-instances are able to make this information explicit and accessible and modifiable [111].

Behavioural reification is the process of converting computation or behaviour into a meta-object [111]. Behavioural reification is determined through reifying method invocations. This reification has access to details of the invocation and can influence behaviours. The behaviours of many systems are implemented in a way that the constituent components of systems are communicating. The communication can be in the form of method invocation in OOP design or message/data passing through a component-based system. Therefore, method invocations or message reification is viewed as a behavioural reification.

In the context of OOP, method invocations reification is viewed as a behavioural reification. In OOP, method invocations are reified into meta-level objects called meta-messages [67, 115]. A meta-message makes the process of invoking a method or sending a message explicit to the meta-level by exposing their reification information. This information is about the implementation and interpretation of the message or method, and can include information such as the caller and callee of method invocation, what is the requested computation, memory allocation and access to method (message) attributes [67, 115].

The method invocations take places at the base-level and reification of method invocations brings these into meta-objects. Reification makes a method invocation into a first class manipulable and computable object (meta-object) and information about the reification (such as origin and destination of method invocation, number of parameters, size of data transport and destination host IP if it is a remote method invocation) is embodied in this meta-object.

In terms of scope, the reification mechanism can structurally and behaviourally reify all or a part of a system in a variety of ways according to the considered design principles in the reflective architecture and MOP. Depending on implementation, reification can be processed during compile or runtime of system. In literature such as [67], the structural model expresses its meta-level with meta-classes, which contain a structural description of base-level objects. In the model, all instances of meta-class have same behaviour. This model has a behavioural reification. The behavioural model makes its meta-level with meta-objects, which are similar to normal object and contain reflective information [67].

There are varieties of models of reification of base-level objects to meta-object(s) in MOP, including one meta-object representing each base-level object (Figure 2.3); a meta-object represents a class of base-level object (Figure 2.4); group reflection (called meta-Computation by [111] explained in section 6.3.2.1) that aggregates the reification of a group of base-level objects into a meta-level object [117]. As mentioned in structural reification, base-level objects are structurally reified in either meta-objects or meta-classes [67, 115] in different model [117]. However, the behavioural reification of base-level objects is handled through interception of method invocations (message passing) of base-level objects and redirecting them to corresponding meta-level objects in meta-level.

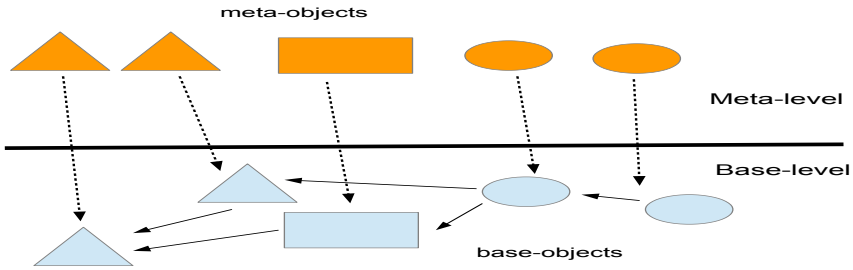


Figure 2.3. One meta-object for each base-level object.

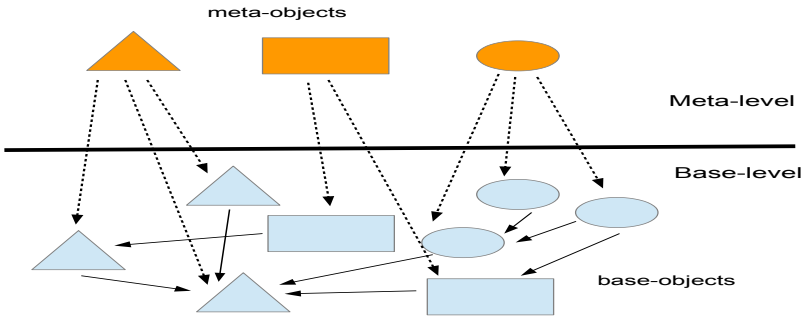


Figure 2.4. Meta-objects for a class of base-level object that reifies.

According to defined design principles of reflective system, the interception of methods, and consequent behavioural reification, can be modelled at the class, method or object level [115].

- Class Reification is a reification strategy for a specific behaviour (characteristic) of all tasks in the system. In this strategy, all methods, which have specific behaviours, need to be redirected to the meta-level.
- Method Reification is a reification strategy that reifies and redirects one or a set of particular method(s) to meta-level instead of redirecting all methods. This strategy is more optimized than class reification, in terms of systems performance, resource consumption and size of produced information in meta-objects.
- Object Reification is a reification strategy that intercepts a set of objects and redirects method invocation for these objects. The object reification strategy is more optimized that class reification or in some cases even the method reification in aforementioned criteria [115].

The reflective system requirements would specify which strategy of behavioural reification is suitable for the system. These strategies could be chosen according to other criteria and limitations such as system performance and resource consumption. For example, if class reflection is chosen in a system with large number of tasks and classes while just the reflection information of individual object is required, such as reflection strategy would result in a large amount of reflection data. The data may not be useful and cause wasted storage, and

computing time, and reduced performance. In the next section, the concept of reflection is explained in the context of programming language.

2.4.1.3 Reflective programming languages

Object-oriented (class-based) programming is one of the most useful approaches incorporating a reflective architecture [61, 62]. In class-based programming, a class specifies a structure for its object in order to respond to methods that are declared or inherited by this object's class. These class-based programming languages are able to introduce a MOP simply through having "class object" structure, which is both changeable and queryable (Class object is reification of a class in a programming language). A class-based language is considered a reflective language, when it satisfies following postulates.

- Every object has a (meta) class that is uniquely associated with it.
- Every object is represented (reified with) by an object.
- Objects are identified by object references and are not empty [61].

In accordance with the reflective language postulates, Maes [62] shows five properties of OOP language (explained in the context of "3-KRS" language) that particularly suitable for expressing reflection and supporting reflective architectures.

- The language should make a separation between objects in base-level and meta-level.
- Every entity in this language is uniformly presented as an object and has a corresponding meta-object. In this language, objects could be: instances, classes, slots, methods, meta-objects, and messages.
- The language should be a complete self-representation. The meta-objects contain fairly comprehensive information about objects. While the contents of meta-objects depend on interpreter design, it is possible to reify (make explicit) other types of object information in the form of a meta-object. In this case, when an object is created, one of the types is used automatically for meta-object creation.
- It should have a consistent self-representation. It needs to have some explicit representation of the interpreter that implements the system. Therefore, when an operation is performed on an object, its meta-object is also requested to perform the action.
- It should be modifiable during runtime. Moreover, this modification impacts on the runtime computation.

In addition, Maes mentioned that 3-KRS Language provides a variety of abstractions for frequently used behaviours. These behaviours can be attached to meta-objects, in case they are needed. In the next section, the mechanism for reflection in the Java programming language is explored. We implement our case studies and all the required applications with Java in this thesis.

2.4.1.4 Reflection mechanism in Java

Java is a class-based programming language, which supports reflection. Java follows all programming language postulates for reflective programming languages. It also follows some of the proposed properties for such systems as specified by Maes [62] (mentioned in previous section) as examined in following paragraphs.

Java differentiates between base-level and meta-level objects and classes. In addition, classes and objects in base-level have corresponding meta-level ones. The Java Reflection API reveals reflection information (meta-information) about methods, constructors, generics (Generics are an abstraction over types and can be used in classes, interfaces, methods and constructors), and annotations of loaded classes. Reflection meta-data provides a complete representation of systems. There are limited methods for modifying objects during runtime in the Java reflection API such as the "newInstance" method in the Constructor class of reflection, setting the "accessible" flag in a reflected object to manipulate objects in a manner that is prohibited.

The Java reflection model provides interfaces, which represent some selected internal parts of a system (introspection). With respect to the point that meta-level objects are class objects and each object in base-level has its corresponding meta-level object, it would be reasonable to say that whenever base-level objects change, corresponding meta-level objects (which have an introspection method) change. If the opposite direction of influence from meta-level to base-level is proved, we can claim that both causal connection and consistent self-representation are proven in this context for the Java reflection API. Smith's requirement [65] (mentioned in section 2.4.2.1) for reflection is for a fully reflective system. Java reflection is facilitated by a complete self-representation feature (explored in earlier this section); however, causal connection is beyond its capabilities.

So, in summary, the Java reflection API provides fairly significant structural reification, and reifies classes, objects, methods, fields, object creation and method invocation, it has limitations on behavioural reification. In addition, the Java reflection API provides an introspection mechanism to present and query particular internal parts of system (self-representation), while there isn't any interception mechanism in Java reflection in order to query and modify the structure and behaviour (specifically the values, meta-data, properties and method's interception) of an object at runtime. These limitations, and the desirability of being causally connected, motivate the idea of developing an explicit MOP, in order to provide a communication protocol for objects and meta-objects.

In terms of application, reflection was initially applied to programming language design [62, 118]. Later on, it was applied to operating system [119] and distributed systems [64, 112, 113]. In the area of programming language design, reflective architecture is primarily designed on procedure-, logic- and rule-based languages (as explored by Maes in [62]); however, Object-Oriented programming languages are more popular owing to their modularity features [61, 62]. Object-Oriented programming languages represent an integration of reflection and objects. In a similar vein, a component-based middleware is very modular and provides an integration of reflection and components [112, 120]. Moreover, it possible to design and implement it on different architectures.

In this section a fundamental principle for designing a reflective and MOP are illustrated in detail, and the reflection mechanism of the Java language is discussed. In this section, the reasons behind the necessity of designing a MOP were explored. In the next section design principles of reflection in distributed systems and especially in the reflective component-based middleware platform will be explained.

2.4.1.5 Reflective component-based middleware platforms

Middleware in distributed platforms makes the heterogeneity of the underlying distributed system transparent from the user and applications perspective and also provides a set of interfaces and services that are usable in any language, operating system or machine environment [66] as shown in Figure 2.5. A middleware such as CORBA [10] and Java RMI, resides between applications and operating systems in a distributed platform to develop,

deploy and manage distributed applications [66, 112]. A middleware usually hides the implementation details from the application (performing as a black-box) [113], because the underlying computing environments of middleware are usually heterogonous and distributed systems with a number of unreliable and unpredictable resources, changes in user and system requirements, and changes in the environment's condition. These changes can trigger huge efforts in redesigning and redeveloping middleware to adapt to upcoming changes [112].

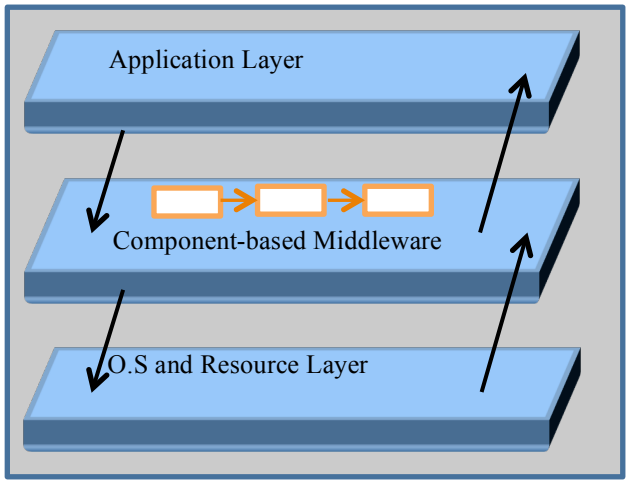


Figure 2.5. Software architecture for Component-based middleware platform.

Middleware requires a new architecture for an open distributed system to adapt to changes and separate concerns. In this architecture, component-based middleware is facilitated by the reflection mechanism shown in Figure 2.6. Therefore, we introduce a new generation of middleware [64, 112, 113] with integration of reflection and component concepts.

A component as a unit for composition and independent deployment is used to construct middleware platforms [112]. The component-based model proposes autonomous and spontaneous cooperation and interoperability among components [64]. Component technology increases reusability and extensibility and also decreases the cost of deployment. In addition, it offers configurability and re-configurability to systems (by adding, removing or replacing components) [120]. While, some middleware (such as CORBA) are constructed with component models that propose a number of functional concerns (such as distribution), runtime component configurability and re-configurability requires a reflective layer to manage and handle these [64, 112, 113].

According to the provided background on reflection and middleware, Coulson [66] and Blair *et al.* [113, 121], the definition of reflective middleware is, "Reflective middleware is simply a middleware system that provides inspection and adaptation of its behaviour through an appropriate causally connected self-representation" [66]. The architecture proposed in [66] is shown in Figure 2.6, provides the separation of concerns in middleware. The base-level middleware takes care of functional concerns, such as distribution, while the reflective middleware is concerned with inspection, composition and configuration of base-level. The design of component-based reflective middleware is explored in the following paragraphs.

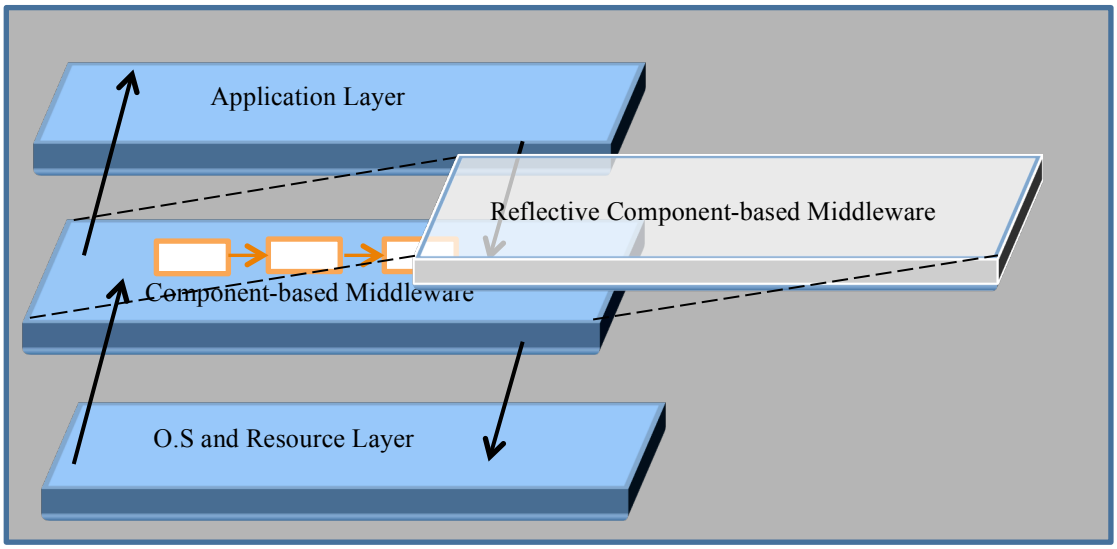


Figure 2.6. Software architecture for reflective Component-based middleware platform.

In the design of reflective middleware proposed in [112, 113], a meta-space is defined to support inspection and adaptation for each component in the underlying infrastructure. The meta-space, as defined in [112], of every component has a number of independent and separate meta-space model. Each meta-space model deals with an aspect (behaviour) of meta-level [112]. Therefore, a component (or object) of the platform (base-level) could have a number of independent meta-objects, in a way that each meta-object coping with one of the defined meta-model [112, 113]. Reflective middleware [112, 113] suggests encapsulation, composition and environment meta-models that are explained below. These meta-models are expressed in OpenORB [116, 121, 122] middleware.

- The *encapsulation meta-model* provides properties related to operations of the interfaces on objects. In addition, it allows access to the representation of an interface in the sense of its set of methods, attributes and key properties of interfaces (such as inheritance structure). The level of access that is provides by encapsulation meta-model is a language dependent matter.

For example. C provides a limited inspection access to the associated interface definition language, while Java or Python provide a broader level of access in terms of delete or add methods and attributes [112, 113].

- The *composition meta-model* provides access to the constituent objects of each object. The constituent objects are a composition of objects that are locally connected together and are shown in terms of object graph. The composition meta-model provides a representation for composite object configuration in a sense of object graph. Therefore, a composition operation would return a graph object with required operations for inspection and adaptation of the composite object. It can be used to provide a structural view of the graph or adapt the structure and content of the graph. Where there is no composite object for the object it returns null [112, 113].
- The *environment meta-model* provides a representation of the execution environment for each interface. It also reifies the engaged processes in the execution of interactions on interfaces. It can be used in distributed environments to provide some functionality such as, dealing with message arrival, enqueing, dispatching, marshalling, thread creation and scheduling [112, 113].

In the reflective middleware in [112, 113] as explained above, each component has a number of associated meta-objects (for instance, three meta-objects as explained above) in a sense that each meta-object is independently responsible for one of the defined meta-models. The meta-object is a reification of a separate aspect of a component that does not interfere with other aspects of the component [112, 113].

A number of reflective middleware prototypes have been developed according to defined principles for new generation of middlewares such as GOPI, OpenORB [116, 121, 122], and MULTE-ORB, TAO and Flexinet [112]. OpenORB [116, 121, 122] is a framework for defining, configuring and also dynamically reconfiguring on middleware platforms to support applications that have dynamic requirements. OpenORB consists of the three meta-models that are defined in this section.

The similarities of middleware platforms and workflow systems in terms of structure of constituent objects (components in particular), dealing with the fluctuation in underlying computing environment and their focus on separate of concerns put forwards the idea of applying reflection concept on workflow system to design reflective workflow systems as

explained in section 2.4.3. In the next section, a number of reflective systems related to the design principles of reflective workflow systems are explained.

2.4.1.6 Reflective Workflow Systems

Reflective systems can be useful in different contexts, from programming [123] to designing high-level distributed applications, middleware [116, 121, 122] and workflow systems [8, 111, 124, 125].

There are a number of workflow systems that are facilitated by reflection. Edmond [124] presents the ROK (reflective object knowledge) reflection model for business workflow adaptivity. It introduces two reflection applications to show the functionality and implementation of a system. The ROK model provides a couple of abilities for workflow about gathering information for system evolution, workflow specification evolution, workflow instance evolution, workflow instance enactment and evolution of process. The Newcastle-Nortel workflow system is a reflective system capable of maintaining and dynamically modifying a workflow system at runtime. This workflow is a decentralized and transactional workflow management system based on CORBA services. It provides a framework for complex service provisioning and meets requirements of interoperability, scalability, flexible task composition, dependability and dynamic reconfiguration [8]. CRISTAL [125-129] (Cooperating Repositories and an Information System for Tracking Assembly Lifecycles) has been developed as the N4U (neuGRID for You) [127] to support the construction of the CMS detector at CERN. CRISTAL is based on a reflective architecture called DDS [125] (Description-Driven System) architecture using meta-objects. DDS handles the complexity of this data-intensive system. DDS like most of the reflective open architecture is divided into a meta-level and base-level. The meta-level contains the description information of the base-level, and the base-levels stores the application data and is based on the OMG multi-layered architecture specification [125, 126]. Neuendorffer [130] presents runtime reconfiguration and parameterized actor-oriented components in hierarchical dataflow models. This model uses a meta-programming tool (named Copernicus) in order to handle runtime reconfiguration. Copernicus operates on the Ptolemy II model [130, 131].

Webb *et al.* [111] developed a meta-object protocol called Enigma (explained in section 6.3.2.1) to introduce grid-related behaviour to PAGIS [111, 132]. Enigma is a meta-level architecture that structurally reifies objects and classes and behaviourally reifies method (messages) invocation. It implements runtime reconfiguration and schedule behaviours. Enigma operates on top a process network model. Webb *et al.* [111] introduced Computation meta-object [111] concept in reflective architecture to simplify the structure of base-level and meta-level in process network applications. Computation meta-object is a reification of a group of base-level objects. It considers the process, process's thread (processes agent) and ports as a single structure and then they are reified into a Computation meta-object. Therefore, the process network model is considered as a network of autonomous computing resources that are connected by unidirectional links. We will use Enigma in chapter 6 and 7.

In general, adaptability, flexibility, configurability, modifiability, and, traceability in runtime are the potential features of systems using reflection. The concept of reflection enables customization of each base-level object (which here means components or processes in workflow systems). In this case, it is possible to reify components (including process and connection) of workflow system and to intercept method invocations between these components. This reflection model over each of workflow system component makes it possible to customize workflow system components individually.

So far, in this chapter, the software techniques (such as MOP) offering adaptability were introduced. In the above sections, we explained the principle concept of reflection, MOP and its application in programming language, middleware and workflow systems. As an early assessment, it seems beneficial to apply reflective mechanism on workflow controller to benefit from reflective workflow system, particularly in some non-functional behaviours such as adaptation, according to previous experiments to design reflective model on dataflow process network [111, 130].

In the remainder of this chapter we briefly survey AOP. The idea of AOP originated from reflective (meta-level) concepts such as meta programming, MOP and reflection system [69, 118, 133]. The AOP and reflective architecture have different mechanisms but they have the same goal of separating business functionalities from technical concerns to improve modularization of programs [69]. Kiczales *et al.* [118] claimed "AOP is a goal, for which

reflection is one powerful tool". Sullivan [134] addressed two deficiencies of MOP, including complexity and overhead (negative impact on performance). AOP proposes a mechanism to solve these deficiencies through cross-cutting concerns [135]. In the next section, concepts and design principles of AOP, as an adaptive mechanism, and AspectJ as an implementation framework for AOP are explored.

2.4.2 Aspect-Oriented Programming Architecture

Most programming paradigms provide a level of modularization and encapsulation of concerns into independent entities with different level of abstractions like, procedures, modules, classes and methods. While some concerns such as security, persistence logging, monitoring and provenance, involve multiple abstractions (cut across a number of entities) in a program, called "cross-cutting" concern. Aspect-Oriented Programming (AOP) is a programming paradigm that provides separation of cross-cutting concerns to increase modularity [69, 70] as we will explore in the next section. AOP enables designers or developers to structure programs (or system) in way that represents the way they think about the programs (or system) [70] and structures their idea [59].

AOP is integrated into existing technologies to provide additional mechanisms to address the cross-cutting concerns in design and implementation. There is a software development methodology named Aspect-Oriented Software Development (AOSD) [136] that is developed based on AOP principles and emphasis on expressing the separation of concerns between functional and non-functional concerns. The functional concerns are mostly referred to business logic of the system. The non-functional concerns in AOP are called aspects while they are known as meta-behaviours in MOP.

2.4.2.1 Principles of Aspect-Oriented Programming

Most programming paradigms provide a level of grouping and encapsulation of concerns into independent entities with different level of abstractions like, procedures, modules, classes and methods. In a procedural programming paradigm, applications are organized based on the required functionalities while object-oriented programming (OOP) suggests grouping related

data and associated methods of an application into coherent entities [69]. However, some concerns are cross-cutting, involved in multiple abstractions in a program. In the next section, some of these programming paradigms are described.

In the object-oriented programming approach, systems are designed by an individual entity (object) and encapsulate entities with "member variable" and "method". The object-oriented programming approach provides a natural way to design complex systems. However, in some cases, all the tasks that should be considered in a design cannot be classified as an object. The object-oriented programming approach also faces with a number of problems, including

- In OOP programming paradigm, methods from an object is invoked from other objects. There is no problem with the way methods are implemented and are modified, because the methods are located in a single class. However, the method invocations are scattered through classes in the application, named *code scattering*, which results in difficult maintenance task and adaptation to the changes in the way that methods are used.
- In OOP programming paradigm, an application is organized into classes that maintain the separation and encapsulation of data and its associated methods into coherent entities. The classes are programmed independently while there are a number of behaviours that may be behaviourally interdependent such as referential integrity rules. The behaviours, named *cross-cutting functionalities (concerns)*, break the independence of classes in OOP [69].

The desire of well-designed system is to keep the system simply maintainable and adjustable, without major structural changes to the system's architecture. AOP application is separated into two application, including *Classes* that is an application that implement the business logic, and *Aspects* that superimposed on classes to address the *cross-cutting concerns and code scattering.* The AOP mechanism separates the expression of concerns and automatically incorporates them into a system. AOP collaborates with existing programming paradigms (and languages) instead of replacing them. These significant features of AOP systems to enhance their expressiveness, maintainability, and utility. AOP is the best approach to express the separation of cross-cutting concerns, while the other concerns easily can be expressed as encapsulated objects, or components. A system can have a number of different cross-cutting concerns (aspects) like, transaction, logging, security, and provenance.

In summary, AOP as a programming paradigm provides separation of cross-cutting concerns to increase modularity. AOP complemented OOP in order to clearly and elegantly develop applications containing code scattering and cross-cutting functionalities. AOP defines a layer

onto the data-driven composition of OOP to handle the integration of cross-cutting functionalities (concerns) through OOP [69].

The reflective architecture is two-level architecture, including base-level and meta-level. The base-level consists of an application while the meta-level consists of supervising and controlling mechanism (Figure 2.2). An AOP application includes functional application and aspects. The functional application is the implementation of the application's business logic in a form of classes or components. Aspects are the programming unit responsible for implementing the application's cross-cuts functionalities (or application's cross-cutting concerns, as defined above), which are also called non-functional (similar to meta-behaviour in reflective architecture). Aspects aim to bring the scattered code throughout the application together to address one purpose [69].

The process of integrating classes or components (representing core functionality) with aspects in order to produce an application is called aspect weaving. Aspect weaving can be carried out during compile or runtime. Compile time weaving is performed by integrating aspects into core program code prior to execution. AspectJ is the most widely used system based on compile-time weaving. It extends classes by aspects and produces an application as a result. The resultant application can be weaved in source code or bytecode. The runtime weaver runs during execution. Examples include Spring AOP, JAC and JBoss AOP. The relationships (binding and removing) between classes and aspects are managed dynamically during execution which is an advantage of runtime weaver [69]. Aspect weaving is usually handled by Java Development IDE, like Eclipse.

There are several concepts in AOP that assist in implementing cross-cutting concerns (functionalities), including joinpoint, pointcut and advices (they are explained in detail in the next section). A joinpoint is a particular point during program execution where one or a number of aspects apply. There are different types of joinpoint has different types including methods, constructors, exceptions and fields. A set of joinpoints is known as a pointcut. The behaviours of an aspect should be applied on pointcuts. These behaviours are defined in advice code. The advices automatically and with help of a Java IDE are woven into the joinpoints defined in the pointcut. There are three types of advice code, namely *before, after* and *around,* that clarify where the advice code should be executed (before, after and both

before and after of joinpoints, respectively). Generally, the implementation of an aspect includes defining the advice code (what the behaviour of the aspect is) and pointcuts (where in the application this behaviour should be applied) [69] as will be shown in Figure 6.8. The definition of AOP concept is similar to AspectJ definition. AspectJ is the first implemented version of AOP. AspectJ is explained in the following section.

2.4.2.2 AspectJ

The concepts of AOP were defined by Gregor Kiczales in 1996, with the first implemented version named AspectJ released by Gregor Kiczales and his team at the Palo Alto Research Center [69]. AspectJ joined the open-source Eclipse community in 2002, which result in developing the AspectJ Development Tools (AJDT) plug-in. This plug-in makes it possible to develop an aspect-oriented program within the IBM Eclipse IDE.

AspectJ, as an extension for Java language, presents AOP concept with following terminology [69, 137],

- *Aspect* is the modularization of a particular concern across multiple classes.
- *Joinpoint* is a particular point during program execution.
- *Advice* is an action at particular joinpoint that is taken by an aspect. There is a number of different types of advice that is listed below,
    - *Before advice* - is executed before joinpoint. It can't prevent execution flow proceeding to the join point, except in throwing exception.
    - *After returning advice* – is executed after a joinpoint normally completed (without throwing an exception).
    - *After throwing advice*- is executed in a condition that throwing exception cause exiting from method.
    - *After (finally) advice* – is executed after a joinpoint exits regardless of normal or exceptional exit.
    - Around advice – is executed before and after a joinpoint.
- *Pointcut* is a set of joinpoints, therefore an associated advice with pointcut will run at joinpoints that are matched by the pointcut.
- *Weaving* is linking an aspect with other application types or objects to create an advised object. Aspect weaver reads the aspect-oriented code and produce object-oriented code with aspects integrated [69, 137].

In this section, the concepts of AOP in AspectJ framework were explained. An adaptive provenance collection mechanism is designed based on AOP (presented in chapter 6) and implemented based on AspectJ concept (presented in chapter 7).

The MOP and AOP adaptive mechanisms designed in workflow controller in workflow architecture provides are capable of inspecting the internal configuration and structure of workflow components and adapt it the different changes in computing environments, users' requirements and underlying system requirements. The workflow controller is an entity in workflow architecture that can be suitable place to apply and manage the adaptation mechanism, because it plays an intermediate role between the entities of SWfMS and computing environment (such as Information Service). The workflow controller has access to required information regarding changes in environment and user requirements. To sum up, adaptive mechanisms (MOP and AOP) employed in workflow controller are able to communicate with an Information Service to realize changes in environments, users and systems' requirements and satisfy many of functional and non-functional requirements of SWfMS. The principles, design issues and implementations of adaptive workflow architecture are elaborated in more detail in the chapter 6 and 7.

2.5 Summary

This chapter has discussed several areas related to provenance, workflow and adaptive systems. While these have largely been treated as separate research areas, we have identified ways in which ideas from each of these areas may be combined in order to provide useful capabilities of provenance and adaptability to scientific workflow systems.

Provenance aims to reproduce, inspect and validate experimental results in the context of scientific workflow systems. It collects, represents and manages required information in sufficient levels of detail for provenance purposes. It is applied to different eScience domains and workflow systems are the primary domain of concern in this thesis. The provenance system and in particular the collection mechanism can be accommodated in platform, framework and application layers in software architectural based on our categorizations.

Workflow systems are categorized into two broad categories of scientific and business workflows. Each of these has a management system enabling specification, execution, reporting, and control of workflows consisting of a number of users, and heterogeneous and distributed resources [10]. In this thesis we focus on scientific workflow systems and their management systems that are more concerned about data and computational composition and execution.

Scientific workflow systems require some components that are responsible for execution and control of communication with the computing environment. In our proposed architecture for scientific workflow systems, the workflow engine and controller are nominated components for execution and control responsibility. The workflow engine aims to run workflow instances based on defined execution rules and scheduling policies. The process of execution needs to model and manage data movement through a workflow system that is handled by the workflow controller. The workflow controller is an intermediate entity between workflow system, user and workflow execution environment.

Adaptive systems react to the changes in the environment and user requirements. Reflection and AOP are two software techniques for designing adaptive systems. We use these two techniques to design adaptive workflow systems. Reflection architecture separates the functional and non-functional concerns of a system into base-level and meta-level layers. The base-level handles the main functional concerns of the system while the meta-level implements the non-functional concerns or meta-behaviours of that system. Reflection can be expressed with MOP. A MOP determines how base-level and meta-level objects can collaborate and communicate through defined reflection and reification mechanisms. The reflection architecture has introduced new features in different context such as middlewares platforms and workflow systems. There are some non-functional concerns (cross-cutting concerns) that are presented more easily in AOP. AOP allows programs to be structured to represent a way designers or developers want to think about the programs [70]. We employ AOP and MOP techniques in a workflow controller that benefits from adaptive provenance collection mechanisms, as explained in chapter 6 and 7.

In the next chapter, we describe a Model of Computation in scientific workflow systems. The design and implementation of an application presenting scientific workflow systems is also described in the next chapter.

3 MODELS OF COMPUTATION IN SCIENTIFIC WORKFLOW SYSTEMS

This chapter aims to investigate how and why a Model of Computation (MoC) explains essential aspects of scientific workflow systems. There are many ways to explain different aspects of computation (for example, lambda calculus is one that models computation at a very basic level). We are going to explain workflow system at an elementary level. Our approach can be characterised by providing a MoC based on dataflow computation. The MoC, in this thesis, is used to explain the underlying operation of workflow and also define how execution semantics are applied when using dataflow semantics.

The explanation of MoC in section 3.1 provides the required background information to describe the semantics of dataflow execution. In this section, a dataflow case study is presented to elaborate how an interpreter applies the semantic of execution model to a dataflow graph. We can define a workflow (for example in the Kepler workflow system) which is essentially a program expressed with a graphical syntax. Our MoC explains how the program operates in terms of the meaning of a graph, which we associate with Kepler. A MoC is a set of rules that determine how the graph executes. An interpreter is a direct encoding of that set of rules.

Section 3.2 explores the significance of dataflow model on scientific workflow systems and particularly on the Kepler workflow system. In this section, the MoC and its relationships to provenance in Kepler are explained. The MoC and interpreter are encapsulated in a Kepler director that contains a scheduling mechanism for taking actors that are ready to fire and then fire them. Kepler has several types of directors corresponding to different MoC's. A director is a supervisory program that implements and expresses the rules of MoC. MoC gives a director its semantics but it is not the direct interpretation of the MoC. Kepler directors take their semantics from an underlying MoC; Kepler supports several different dataflow oriented MoCs, including Synchronous Dataflow (SDF), Dynamic Dataflow (DDF) and Process Networks (PN). In this section, the provenance mechanism including collection mechanism is explored and architecture is presented to provide some background for provenance architecture. Provenance in Kepler is handled by a provenance recorder actor. We will propose alternative provenance architecture in chapter 6.

Section 3.3 investigates semantics of execution in Process Network (PN) dataflow. PN is a deterministic MoC for scientific workflow that is used in different workflow systems, particularly in the form of a director in the Kepler system. We explore PN as one variant of dataflow representing concepts of scientific workflow systems.

The information regarding MoC and process networks in this chapter provides enough understanding to design and implement a process network application, as explained in section 3.4. The process network application can be seen to express the principles of a scientific workflow system, and will be used in design and implementation of adaptive provenance collection mechanism for scientific workflow systems in chapter 6 and 7 in this thesis.

3.1 Scientific Workflow systems with underlying model of computation

In this thesis, the execution models (referred to MoC) explain the underlying operation on workflow (or dataflow) and also define how execution semantics are applied on a workflow [30]. The semantics expressed by a MoC in a dataflow computation is a set of rules governing the node interactions that can be expressed with an interpreter. An interpreter is an entity that applies a MoC on a dataflow graph. An interpreter can be considered as a supervisory program that knows how to deal with input data, how to process input data and produce output data. Thus, we can use interpreter to answer questions about execution, such as, when does a node start processing.

The answer reflects interpreter‘s semantics, such as start processing when received all the input data; start processing with some of the input data (introducing the notion of scheduling policy); or even start processing when a request for data or process is received from a downstream node (a demand-driven scheduling policy). This is a primitive way to look at a workflow and workflow execution, but useful because it gives fundamental understanding.

In the following section, a simple dataflow model is presented in order to exemplify a number of fundamental dataflow interpretation concepts (including how graphs are executed according to given MoC, a mechanism of selecting the next node for execution, a mechanism of graph node activation, a mechanism for checking input data availability, the order of

execution), and establish an area of discussion to describe different Models of Provenance (presented in chapter 4) in the context of scientific workflow systems.

3.1.1 Case Study: A Simple Dataflow experiment

Nodes (processes or actors) in dataflow graphs represent one or a number of tasks. The task defines an atomic (functional) computation task. When a task fires, it applies this computation on sequence of inputs and delivers the result as an output sequence. The functional behaviour makes it possible to map input data into output one free from side effects. It means that the output data is a function of input data. Nodes in dataflow networks contain a number of repeated firings of dataflow actors. There are a set of firing rules that specifies exactly which sort of input should be available to fire a task [138].

In this section, a simple graph is shown in Figure 3.1.a to describe how a simple and conceptual interpreter encodes the rules of MoC, which defines the semantics of that model on a graph that it can be specialised for scientific workflow systems such as Kepler and Ptolemy (explained in section 3.2). In this section, we provide some required background to describe the primary concept of model of provenance in the next section.

A dataflow graph [27, 139] is presented in Figure 3.1.a that consists of five nodes. The operation of nodes is indicated in each node and also presented in a form of mathematical model. The first node (n1) adds two inputs, the second node (n2) just copies the input data into two separate channels linked to node 3 (n3) and node 4 (n4). The node n3 and node n4 are operating (multiplying and subtracting respectively) on input data coming from n2 and constant value input that is provided for them. The node numbered 5 (n5) adds two input data and put the result into output. Channels connect nodes in this dataflow graph. A channel can be a queue or buffer data structure to transfer data. In the simple graph, it is possible to consider a variable as a channel data structure because just one item is transferred (a single data structure) each time rather than a sequence of data.

The following set of rules is considered as semantics for the execution of the graph shown in Figure 3.1.a.

- Input data would enable a node and makes it ready to fire.

- The existence of data in node makes it fireable (node is placed in fireable queue shown in Figure 3.1.b).
- Output data is produced immediately by the operation in node as soon as input data is available.
- The output data is sent to the succeeding node (or nodes).

In this example the input data are assumed to be atomic data structures (not complex data structure such as nested data collections). A single operation is applied on input data with a single method invocation (operation does not need multiple task invocations). A single node is executed at a time with one execution thread. If more than one node is ready to fire (they are in fireable queue (see below)), the first one in queue is selected and served (FIFO policy for dequeuing).

In this simple dataflow graph, the interpreter checks for the availability of value in input ports. If the required input values were available in input port, the corresponding node would be placed in fireable queue (a queue contains nodes that are ready and waiting for execution (fire)). The nodes, in fireable queue, are fired according to their order in queue. There is just one task in each node and just one operation is defined in each task. Moreover, task is fired once for each set of input values.

In the example in Figure 3.1.a, the graph execution is triggered as two inputs entered into the input ports of n1. The node n1 checks its input ports; it is ready to fire (added into fireable queue) as two input values are available. The node n1 performs a sum operation on inputs and the result is delivered to the output port. The output value from n1 is written into the output channel to be transferred into the input port of n2 immediately (because it is non-blocking writing), which causes n2 to be added into fireable queue. The node number two (n2) is fired as input value is received, the delivered input valued into two output ports. The n3 and n4 nodes receive the generated values by n2. They need another input value, named constant value input, as shown in the Figure 3.1.a. Suppose that those constant value inputs have already been initiated.

When the execution of n2 is finished and values are transferred to n3 and n4, both nodes are ready to fire. In order to solve the scheduling issue of which order to use, it seems reasonable that the interpreter define a list of nodes that are ready for execution (fireable list), as shown

in Figure 3.1.b. Then according to interpreter scheduling policy, one of the nodes from fireable list is selected. We implement this list by a FIFO queue. This queue is used as graph execution is started. Therefore, in this graph when n2 transfer value into n3 input port. Then n3 is added to fireable queue (shown in step 3 in Figure 3.1.b), if another input value is available (the similar scenario happens for n4). There are two nodes (n3 and n4) into fireable queue (shown in step 3 in Figure 3.1.b). Our FIFO interpreter scheduler selects n3 and fires it, then transfer n3 output into n5 input port. The node n5 is not added to fireable queue yet, because it expects another input value from n4. The interpreter will select another node (n4) form fireable queue (shown in step 4 in Figure 3.1.b), if it is not empty. The node n5 is added to fireable queue (shown in step 4 in Figure 3.1.b), when the n4 is fired and its result is transferred into n5 input port.

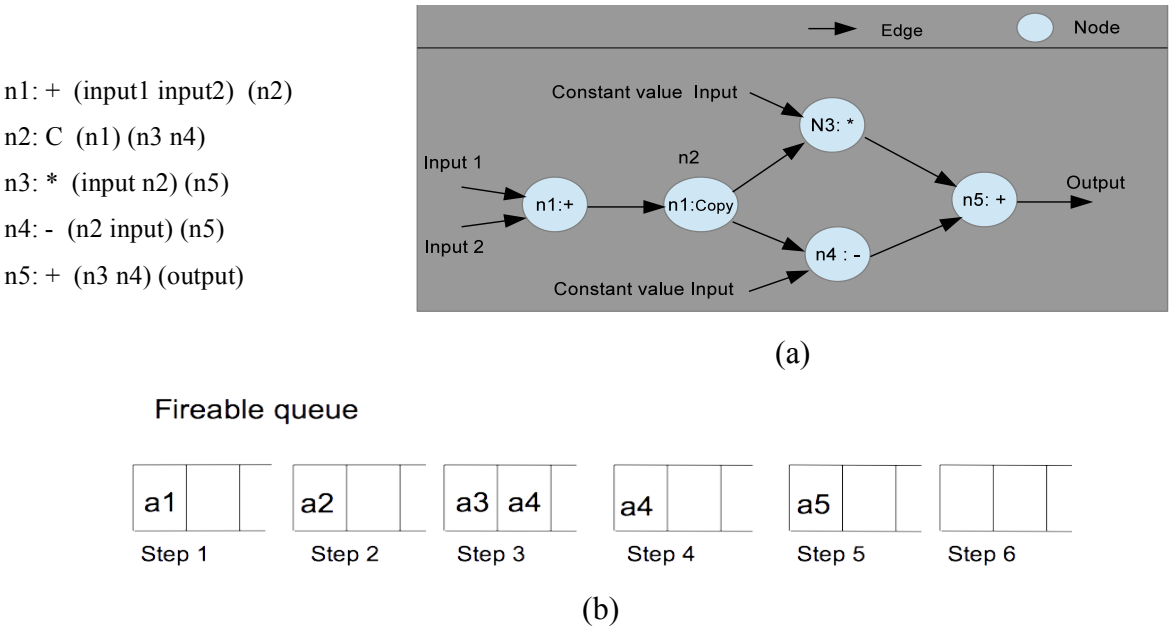


Figure 3.1. (a) a simple dataflow; (b) Fireable queue in Interpreter.

In this section, the concept of MoC in dataflow and scientific workflow systems was explained in the context of a simple case study that shows how an interpreter applies MoC on a dataflow graph. In the next section, we look at Kepler system as a variant of scientific workflow systems, to explore the significance of dataflow model in scientific workflow systems.

3.2 Influences of dataflow model on important workflow systems

The dataflow MoC underlies models used in a wide range of computer systems from embedded processing and signal-processing architecture to distributed computing workflow. In many dataflow models, there is a discipline for transferring data among nodes (processes or actors) that guarantees data arrive in a correct order. As nodes do not share their state in dataflow model and also nodes communicate through ports, which enable efficient concurrent (parallel) and sequential model implementation [130, 138-140].

As explained in the previous section, it is necessary for an interpreter to execute the graph network (or in more general terms execute the workflow system) with the help of defined execution semantics in the MoC. There are wide variations in the applications, topologies and semantics of executions that are expressed by an interpreter. Consequently, workflow systems (such as Kepler and Ptolemy II) can benefit from having a variety of MoCs to cater for scientist requirements of conveniently using and designing scientific workflow systems.

This section provides background information to demonstrate the construction of underlying workflow systems, providing enough understanding to design workflow and dataflow applications. In the next section, Kepler and Ptolemy II are introduced and their MoC are investigated. Following that in section 3.2.2, the mechanism of provenance collection in Kepler and its actor are explored.

3.2.1 The MoC in PtolemyII and Kepler

Kepler [28] is a scientific workflow system that can be used in a wide range of disciplines. Kepler is designed to be used by not only computer programmers, but also by scientists to create, execute, and share models, and by analysts to analyse in different science disciplines [141]. It supports different types of workflow ranging from local analytical pipelines to distributed, high–performance and high-throughput applications, which could be data- and computational-intensive [22]. Kepler is coordinated by a component named the director. A director is a supervisory program that is consistent with the rules of MoC and implements the rules of MoC. Kepler is built based on Ptolemy II system [30] and Kepler's director inherits

the semantics of a variety of Ptolemy II‘s directors. The Kepler workflow system is explained in detail in chapter 5.

Ptolemy II generally builds models based on existing and independent actors (that are similar to the process and node that is called actor in the context of Kepler and Ptolemy II systems) that usually represent operations or data sources. The semantics of composed workflow models are controlled by the director [142]. Ptolemy II is an actor-oriented modelling and design tool. It is facilitated by atomic and composite actors, communication ports, relations that connect the channels between ports and link between a port and a relation create the communication channels between ports [131].

Actors in Ptolemy II and Kepler contain six action methods, including namely “Preinitialize”, “Initialize”, “Prefire”, “Fire”, “Prefire” and “WrapUp”. The “Preinitialize” method runs once per execution before of others methods to set up port type and scheduling information. The “Initialize” method initialises the parameters to make actors ready for execution. The “Prefire” method checks firing rules and decides whether the actor is ready or run or not. The “Fire” method performs the main functionality and computation of the actor. The “Postfire” method schedules the next firing and also updates the actor state after the completion of actor execution. The “WrapUp” method invokes once at the end of per workflow execution, in order to display final result [108]. Directors use a MoC to encode the execution of actors in a workflow through invoking the methods inside actors (see following for detail explanation).

Ptolemy II supports several different views of system modelling and each of them is used for specific purposes. Ptolemy has variety of directors that are suited for different models dealing with concurrency and time. Choosing an appropriate director has a significant effect on the design quality of the system [30]. A director takes a MoC, which is used to encode workflow execution. It a form of scheduling mechanism using firing rules determined in MoC for executing workflow

The Ptolemy and Kepler domains contain many directors [29, 30], but the ones of most concern here and closest to scientific workflow systems are Synchronous Dataflow (SDF), Dynamic Dataflow (DDF) and Process Networks (PN) [29, 30, 108]. The domain that is of interest in this thesis is Process Network (PN) (explained in detail in section 3.3).

Synchronous dataflow (SDF) domain models simple dataflow, which is not concerned with complex flow of control [29]. In the SDF domain, a fixed number of tokens per firing is produced and consumed by actors, [30]. The SDF is a restricted dataflow model with fixed token rate during execution, finite sequence of firing (the number of firing rates is pre-determined), and use bounded memory [130]. In addition, it schedules the firing of actors (order of execution) statically, which make it possible to have a compile-time scheduling. It executes just a single actor at a time with one execution thread [30, 108].

Dynamic dataflow (DDF) domain is a superset of SDF. The DDF scheduler dynamically schedules the firing of actors. The rate of production and consumption of tokens by actors could be changed in each firing (iteration). DDF schedules the actor execution iteratively by searching for ready actors which lead to workflow or a part of that are executed repeatedly or conditionally [30, 108]. It offers control structures such as loop and branch to workflow system but it is restricted in parallel processing (for which PN is suitable as we will explore in section 3.3).

Kepler inherits the semantics of a number of Ptolemy II directors. Kepler distinguishes between the workflow graph and the MoC by specifying an appropriate director for each workflow. A MoC states all inter-actor communication behaviours [143]. Kepler execution mechanism could be considered as an orchestration in which the director is a central management point. The main activities of a director are:

- It invokes all actors' pre-initialize methods once a time before the workflow execution.
- It checks the type of all connections and ports.
- In each run, it invokes the "initialize" methods to pass information into next actor.
- It runs the "fire" methods in which the major functionalities are implemented. An actor runs usually includes multiple iterations. In each iteration, "prefire", "fire" and "postfire" methods are called. It performs the following sequence of activities (the asterisk denotes activities that may be executed zero or several times during each workflow execution) [30]:

  Preinitialize → (initialize, (prefire, fire*, postfire)*)* → WrapUp.

In this section, we explored the concept of MoC in Kepler. In the next section, the mechanism and components of a provenance collection mechanism in the Kepler scientific workflow system are explained. The next section provides an understanding of the concept of

provenance in workflow systems. We will return to the explanation of concept of MoC in section 3.3.

3.2.2 Provenance in the Kepler workflow system

There are varieties of approaches to collect provenance information in scientific workflow systems, as we will explain in chapter 5. The collection mechanisms mentioned in chapter 5 have different design dimensions presented and examined through a number of workflow systems. Kepler is also one the scientific workflow systems considered in section 5.2.1. We are now going to explore the provenance component of Kepler, as one variant of scientific workflow systems, in this section, to provide an overview of provenance information and provenance collection architecture.

Kepler's provenance component (Provenance Recorder) is added to workflow instance of Kepler workflow architecture, as it is shown by blue-square in Figure 3.2 and Figure 3.3. For example, provenance schematic view in Kepler provides an overview of how workflow systems handle their provenance collection mechanism. We design our reflective provenance architecture, presented in chapter 6, as alternative approach to extend the capability of provenance collection mechanisms.

In the Kepler schematic view, a Provenance Recorder component activates the provenance collection mechanism and collects provenance information. Kepler's provenance component is optional, depending on the user's requirement to track provenance in a workflow instance. The Kepler Provenance Recorder provides an extensible framework for capturing provenance information in the Kepler workflow system. Provenance recorder records provenance information regarding input and output data, their metadata, the context of workflows, specification of entities, intermediate produced data, evolution and execution information [46].

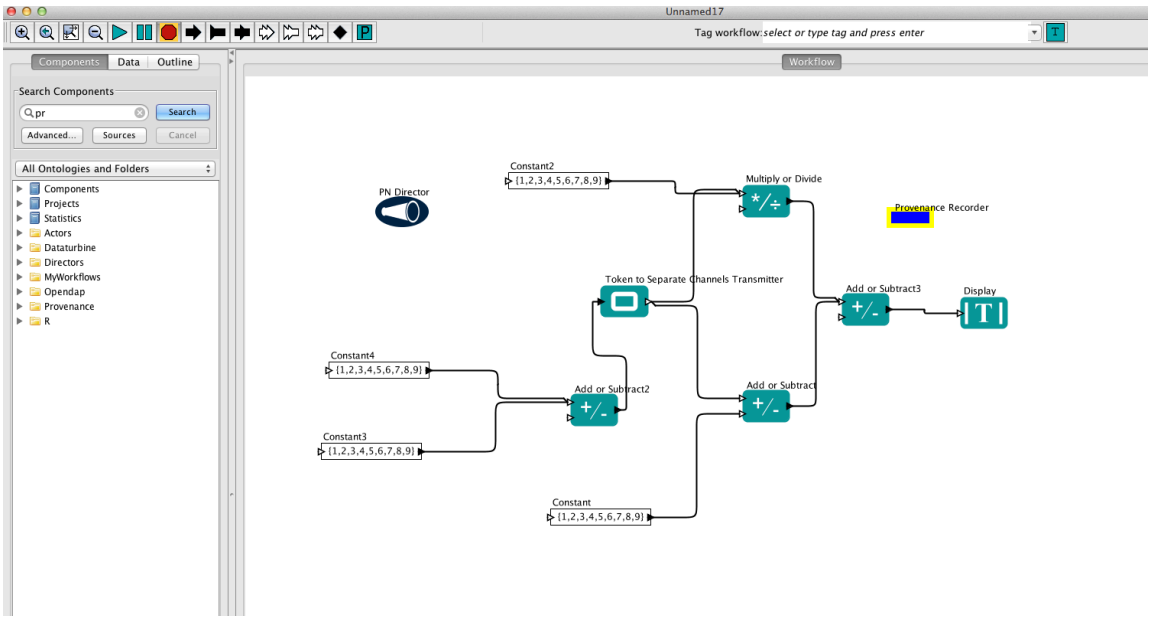


Figure 3.2. A workflow example in the Kepler workflow system executed under supervision of PN director and facilitated by Kepler's Provenance Recorder.

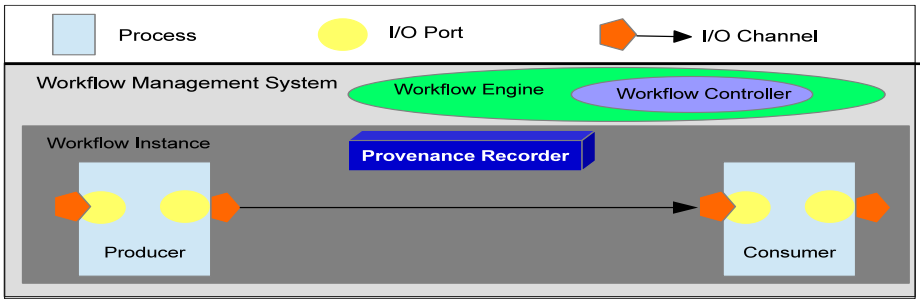


Figure 3.3. Kepler provenance schematic view.

The Provenance Recorder captures provenance for workflow specification, and execution phases. The Provenance Recorder can only operate on workflow instances using one of SDF, DDF or PN directors. It can store provenance information in a text file and relational database (also in XML format). In Appendix F, we explore how to configure Provenance Recorder in source code, and show this choice of storage format. Kepler includes a relational database schema in order to store all necessary data into database tables. This design makes it possible for Kepler to cover required provenance information regarding workflow components to reproduce and re-run workflow. This relational schema can capture three types of provenance information [46, 144].

- *Workflow specification*: The information regarding the context or specification of actors, directors, parameters, and ports in each workflow

- *Workflow evolution*: This information is related to the changes in workflow components specification during the workflow lifetime.
- *Workflow execution*: The information that is generated and recorded during execution.

The workflow specification information of workflow is collected when the workflow components are initiated. The provenance recorder collects workflow specification information, when the "preinitialize" and "Initialize" action methods are finished. The information is stored once in database for workflow executions, even if this workflow runs several times. If any changes happen in the specification of workflow components and workflow configuration (such as changes in actors or in parameters' value), these changes are recorded as information about workflow evolution. The workflow execution information is collected and stored in database by the "postfire" methods, because the actor's state is updated and persistent.

There are four different types of events that may happen in during Kepler during workflow execution. The Provenance Recorder collects provenance information from these events, as explained below. The "FiringEvent" and "IOPortEvent" are mostly concerned in Kepler, so fully implemented, in contrast with "ProvenanceEvent" that provide a room for user to customize and implement the events [144].

- FiringEvent: An event published by directors whenever an actor is activated (including the "prefire", "fire", or "postfire" action method in actors).
- IOPortEvent: An event published by an input and output port (IOPort) when a token or tokens are sent or received. In Kepler, the provenance recorder use these events to save intermediate results of the workflow.
- ProvenanceEvent: An event that is published by actors when they fire in order to report provenance. Any actor that wants to record extra provenance about how it is carrying out its job, can create and publish this event.
- IOPortRefillEvent: An event published when a port is refilled.

In this section, Kepler's Provenance Recorder actor and its architecture was discussed. We will discuss other possible workflow architectures for provenance collection mechanisms in chapter 6. As mentioned in this section, PN is one of the Kepler directors that is of interest in this thesis. In the next section, process network is explained in more detail as a variant for scientific workflow systems.

3.3 The Process Network model as a foundation of workflow

In this section, we explore PN's MoC in general and as a director in the context of the Kepler workflow system. We will describe the ability of PN dataflow to represent workflow systems, in order to provide enough understanding to design a process network application. We then use that PN application as a vehicle for exploring our ideas in workflow architecture and provenance collection.

As mentioned in section 3.2.1, Kepler has a number of directors through which we explored the principles of SDF and DDF. Process Network (PN) is another MoC represented as director in Kepler. Kahn [145] in 1974 put forward underlying semantics of PN in terms of generalized process with communication via asynchronous communication (implemented in put and get methods) over unbounded FIFO queues. PN can be implemented in several ways including [30, 111, 146, 147].

Process networks (PN) use asynchronous token passing as their fundamental communication mechanism to pass tokens between sender and receiver actors through unidirectional and unbounded channels. In this model, actors map input token sequences to output token sequences. The receiver actor receives the incoming tokens from a FIFO channel.

The way components communicate is defined in the communication model [30]. The communication model in PN is asynchronous message passing, which token is regarded as message in this model. Nodes communicate by sending tokens through queues (unidirectional FIFO channels). The sender node (producer) sends tokens and concurrently continues its work without waiting for token to be received in receiver (non-blocking writing), while the consumer (receiver node) reads from a blocked channel when data becomes available. Moreover, tokens in channel‘s queue are delivered in order of transmission with the promise of lossless transmission by the queue. The communication model in PN occurs through ports, which it makes the processes insulated and transparent, which enables efficient concurrent and sequential model implementations [130]. In this model, nodes map input sequences of tokens to output and there is a conceptually infinite channel (bound by physical resource limitations) carrying sequences of tokens between entities. Half-Channel (shown in Figure

3.7) [111, 132] is another way of achieving asynchronous communication which is explained in section 3.4.

In the implementation of PN as director of Kepler [29, 108], a thread (process thread) is assigned to each actor, and run according to data availability [29, 108]. The sender actor passes tokens to receiver actors through unidirectional FIFO channels. The receiver actor is blocked when trying to read from an empty channel. A process thread checks the availability of tokens in channels frequently through its "get" methods. As tokens become available in channels (through input ports), the actor is fired and produces output tokens. Then the actor checks channel capacity and puts produced output tokens into channels (through the "put" method of output ports). The sender actors are blocked when the channel capacity is full and should wait for free space in the channel [29].

In the simple dataflow graph shown in Figure 3.1.a, the semantic of firing rules in DDF MoC was explored. We explain the semantics of PN in Figure 3.4, which is a simple PN dataflow. All nodes in PN run simultaneously, and flow between them is regulated by the "put" and "get" methods. A process thread is assigned to each node. Nodes frequently check the availability of tokens in channels. The process thread of a node runs the "get" method of each input port, operating on the associated input channel, to get the input token. The existence of tokens at input ports of a node makes it fireable. The node produces output tokens and sends them immediately to an output port (all output ports if they are more than one). Each output port has a "put" method that puts the produced token(s) into the associated output channel.

In this simple PN dataflow graph, the availability of value at an input port is checked with the "get" method. There is just one task in each node and just one operation is defined in each task, thus task is fired once. The graph execution is triggered as two inputs entered into n1 input ports. The "get" methods of input ports of n1 check its corresponding input channels. The n1 node accesses to the provided input values by the "get" method in input port, and then the sum operation is performed on inputs and result is delivered to output port. The output ports of n1 (with the help of the "put" method) write output value into output channel to be transferred into n2 input port. Node number two (n2) is fired as the "get" method of input port receive input value. Then it delivers an input value into two output ports (because it is a copy node). The node n3 and n4 receive the generated values by n2. They need another input

values (named constant value input). Suppose that we have already initiated by constant value input during graph construction.

When n2 execution is finished and values are transferred to n3 and n4, both nodes are ready to fire. There was scheduling issue in simple dataflow that was solved by a fireable queue in interpreter (shown in Figure 3.1.b). The multi-thread implementation in PN solves the scheduling issue because the nodes can be fired simultaneously and independently. This is one of the main difference between semantic of PN and SDF (that is encoded in simple dataflow in Figure 3.1.a). The n3 and n4 are fired simultaneously and write their output in their output channels whenever they accomplished their firing process. The "get" method of input port of n5 is blocked until receive both tokens from n3 and n4. Finally, output value, which is written to output with the "put" method.

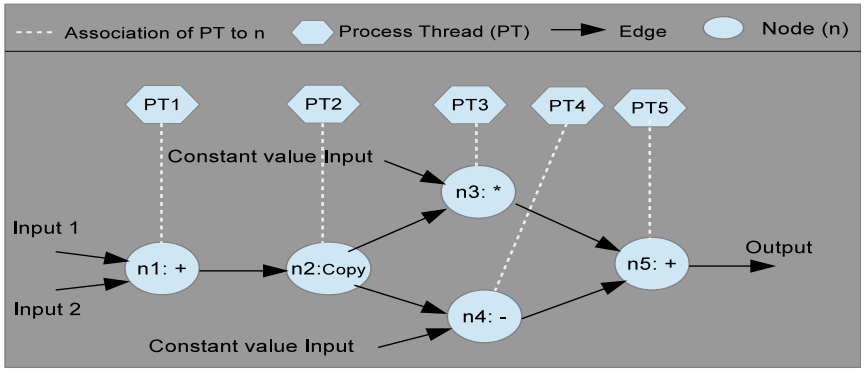


Figure 3.4. A simple PN dataflow.

PN model is powerful in comparison with most MoC, including SDF and DDF as defined in Ptolemy and Kepler, because of having independent processes that are run simultaneously (owing to using multiple thread) [29]. Therefore, PN is an appropriate model for parallel and distributed systems [30, 108]. As it is shown in Figure 3.5, PN model contains SDF and DDF models as sub-domains [148]. The PN director is not pre-calculated scheduling model, in contrast to SDF. Thus, the number of firing execution in PN model (unlike SDF,) is not determined. The DDF semantic is closely related to PN semantic, except in their approach to deal with input data (initializer). The PN actor is blocked until it receives data (from its inputs), while the DDF actors use firing rules to determine the invocation condition (initiating and executing actors) [29]. Generally, PN can be considered as a superset model of DDF [145, 148].

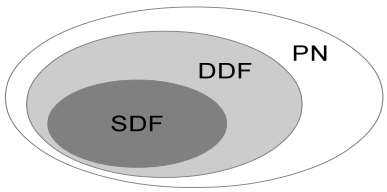


Figure 3.5. The hierarchy of Ptolemy dataflow models [148].

PN is a deterministic MoC for dataflow (scientific workflow) system which can have either demand-driven or dataflow [147] scheduling policy. A demand-driven scheduling policy runs (fires) processes only when a successor process requires tokens; a dataflow scheduling policy runs (fires) processes when the input values of process are available (independent of demand).

We now set out to show that the study of properties of PN systems will give us insight into properties of scientific workflow system (dataflow). We justify this belief by observing that PN is recognised in Kepler workflow system as a powerful sub-domain and many workflows are expressed in terms of PN. PN director as one of the directors provided in Kepler workflow systems is evidence for considering PN model as an adequate representative of scientific workflow systems, as it is shown in Figure 3.2. As noted above, domains such as DDF and SDF are a sub-domain of PN and could also be expressed in terms of PN.

In this chapter up to now, we have explained the concept of MoC and the way it applies and influence on dataflow. Following that Kepler as a scientific workflow system was introduced to make the concept of MoC and interpreter clear in this context of workflow system. In this section, we have presented PN as an adequate representative for scientific workflow systems. Therefore, we use PN dataflow as the foundation of workflow in this thesis. In the next section, we describe a design and implementation of a process network framework that would be used in later chapters in our implementation of adaptive provenance collection mechanisms on scientific workflow systems.

3.4 Process Network Application

In this section, design principles of a framework for PN are introduced in the context of our Process Network Application (PNA). PNA has an abstract PN MoC that is facilitated by port-

centric, half-Channel [111, 132] (see below) and token data types. In this section, we show two case studies implemented in PNA.

PNA is a component-based process network application aiming to develop an architecture utilized by a high-level MoC to construct an adequate representation of workflow system (refer to previous section). It provides a framework for implementing different provenance capturing mechanisms. In the rest of this thesis, it is coupled with AspectJ and Enigma [111] in order to implement Aspect-Oriented programming and MetaObject programming approaches for collecting provenance information (presented in chapter 7). In the PNA, computational resources, called processes, are connected with communication channels. The design principles of PNA are inspired by a number of concepts and frameworks such as PN [146], PAGIS [111, 132, 146], Ptolemy II [29, 30, 138, 149, 150]. The PNA communication architecture is a port-centric model [132, 138] and benefitted from Half-Channel concept of PAGIS [132]. In addition, the data types that is conveyed through the communication architecture is named token [150] which is a concept defined in Ptolemy II [150, 151].

PNA can be considered as a computational engine that handles a computational model. The computational model abstracts the underlying computer language and makes the complexity of real machine transparent from a programmers' point of view to improve their understanding of the system and program performance characteristics. PNA as a computational engine that executes processes and coordinates communication with its implementation structures which are mostly implemented based on [146], as elaborated below. PNA hides implementation decisions such as deployment and scheduling. PNA is able to execute process networks under a dataflow and demand-driven scheduling strategy, as defined.

PNA has a number of mechanisms, such as factory and builder, to construct elements of process network (as shown in Figure 3.6) including process, channel, and some mechanisms to tie elements together (building a process network). The mechanisms are not the core principles to network execution but they are important for network reconfiguration. PNA is a process network framework containing a number of processes that communicate by sending token data structure between actors. The order and policy of process execution in PNA is defined in a scheduling policy. The core design structure of PNA (including factory, builder

and framework classes) and some components of PNA, such as half-Channel, scheduling, structure of actors, and threads, are based on PAGIS. However, we made significant enhancements changes to the PAGIS application

The version of PAGIS that we used is constructed and initiated from meta-level. PAGIS can only operate when is integrated with Enigma, explained in section 7.2.1 (and only when Enigma is operated with the notion of meta-Computation). Meta-Computation [111] (which is expressed in section 2.4.1.6 and will be explored in section 6.3.2.1) is a complex representation of constituent objects of PN in meta-level to simplify the representation of PN and also to customize all the constituent objects effectively. However, it has a deficiency in the customisation of each object of PN.

- We made the execution of PNA Independent of Enigma. We need a PN framework to work independently in order to integrate it with AspectJ to implement AOP oriented provenance collection mechanism. We made some changes in PAGIS design and add new design (such as defining new factory classes named "PNAKahnFactory") to PNA design in order to make its run independently from Enigma.
- We represented the PN with a primitive representation that consists of processes, channels and ports. The representation of PN defined in this chapter and is shown in Figure 3.6.
- We implemented the "Token" data type in PNA, so we could claim that our framework is independent from data type and could work with different data types like, stream data, image, and array. We implement our case study that works with integer while the extension of token enables our case study to work with other data types similarly, without any changes. All the data types are considered as token data type (implementation of PAGIS does not utilised token). As we will see later (when we consider the generic Open Provenance Model (OPM) in section 4.4), we need a particular data type. The used data type (in Open Provenance Model) needs to be presented as a uniform type to make the data type transparent from workflow execution and model. Therefore, we could present undefined, complex and structured data model in our provenance system (model of provenance).
- We implemented a number of Actors (processes) and classes to implement our case study. The actors include "ArithmeticProcess" (which gets the mathematic operation and input data and return results), "IntegerProducer" (which generates in defined range of integer tokens), "PNAKahnFactory" (explained later in this section) and "PNAMOPFActory" (explained in chapter 6).
- We made some changes in methods and classes to make them compatible with our design for adaptive provenance collection mechanisms.

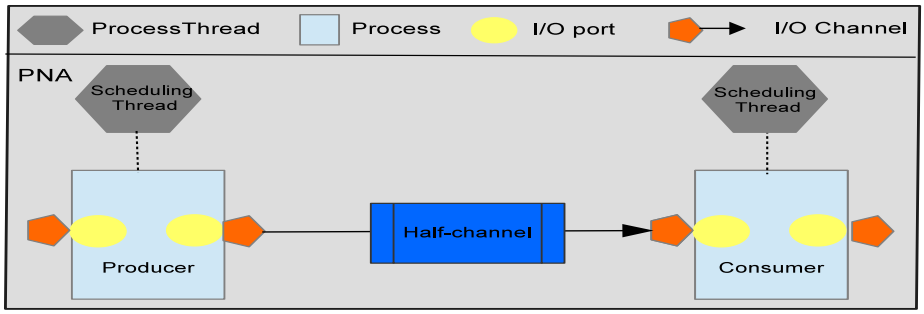


Figure 3.6. PNA producer and Consumer process network.

PN is the integration of processes and channels. “Framework” class provides safe environment for constructing a PN. It separates the specification of structure of PN from the required actions to realize it. The “Network”, “Builder” and “Factory” design patterns help “Framework” supporting this separation. The aggregation of processes and channels that are connected them together makes a network that is handled in “Network” class. “Builder” class is defined to separate the construction of objects from their representation. The “Builder” describes the creation events of processes, channels and ports; and also checks the construction of PN.

PN needs mechanisms to construct the elements of PN. Factory is one of the mechanisms employed in PNA to create PN constituent components (and also it is extended to construct meta-level). Three factory classes are implemented in the PNA application, named “PNAKahnFactory”, “PNAMOPFactory” and “PNACompFactory”. The “PNAKahnFactory” assists in creating entire PNA components in the PN including, network, builder, framework, process, channel, input port and output port. PNA with the “PNAKahnFactory” can run standalone without any dependency to meta-level. However, “PNAMOPFactory” and “PNACompFactory” (explained in section 7.2.1) are coupled with Enigma to construct both base-level and meta-level, explored in section 7.2.

PNA process has a number of methods, including “prefire”, “isFireable”, “fire”, “getData” and “updataData”, which are responsible for initializing, applying transforming and managing communication with input and output ports. The “isFireable” method decides about next execution of “fire” method. The main functionality and transformations in processes in PNA API are modelled in the “fire” method, while the execution of the “fire” method is determined in the “prefire” method. There are a number of properties inside processes that might be set or

updated in the “prefire”, including “initial properties” and “runtime properties” that are initial configuration of the process before execution and are modified properties of the process during execution.

PNA API uses a port-centric [111, 132, 138, 149] communication model in which channels are defined by the ports that connect a channel to processes. PNA provides a communication mechanism that enables to send tokens in distributed environment through channels. In the centralized model producer writes its tokens into channel, and then consumer reads the tokens. The process of read and write does not have latency cost, because it is performed on the local channel storage. In a distributed model, if the channel is located in either of producer or consumer sides, it causes a latency cost, which it has an adversely effect on performance.

The Half-Channel (shown in Figure 3.7) [111, 132] is proposed as an effective way of achieving asynchronous communication. It has a particular advantage in localizing process behaviours, and putting boundaries around processes to facilitate distributed reconfiguration (as shown in [132]). It uses a pair of producer-consumer buffers and a transfer thread that transfers tokens from the producer channel to the consumer channel. The half-channel design insulates a process from its neighbours and networks. In addition, it makes the communication transparent for processes in distributed environments. It hides distribution issues without an unfavourable performance effect. This insulation makes the process of reconfiguration simple. The half-channel provides a quiescence state, so it would be possible to have safe (insertion, remove and replacement) reconfiguration in the structure of a PN.

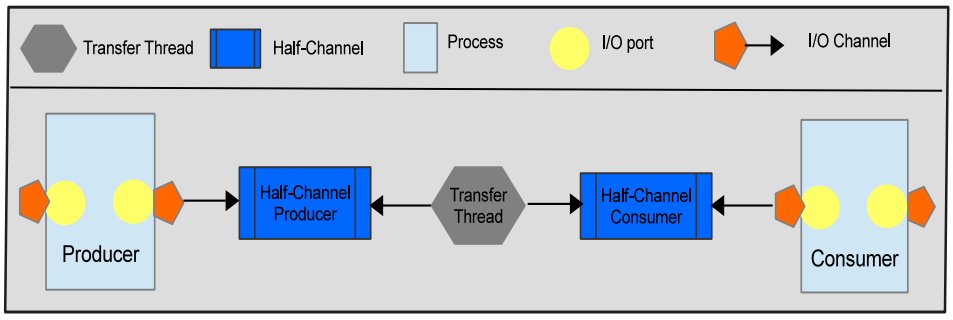


Figure 3.7. Half-channel Design.

The token type provides a standard interface to encapsulate application data aiming to uniformly handle data without concern for data structure. The encapsulation of data makes it

possible for developer to extend the library of data types that are defined in Ptolemy II. The data encapsulation and its derived classes are called Token. This enables interaction of the application with user interfaces without detailed prior knowledge of the data structures [150, 151]. We implement a number of data types (including Base, Object, Actor and Structured type) and a number of tokens that are defined based on defined data types (such as "IntToken", "ScalarToken", "StringToken", "BooleanToken", "ActorToken" and "UnsignedByteToken").

Scheduling of processes is one of the main considerations of any PNA. PNA assigns and implements a Java thread for each process named "Process Thread" (shown in Figure 3.6) that invokes methods of a process based on the defined scheduling strategy. Therefore, the assigned "Process Thread" to each process frequently runs according to implemented strategy. The scheduling strategies in PNA are dataflow and demand-driven. The dataflow strategy would fire to the capacity of the channels while demand-driven strategy would fire to demand from output channels. The demand in the demand-driven strategy is initiated from consumer processes, so as a demand is received, the process is fired.

The run method of "Process Thread" is presented in Figure 3.8. In this method, thread invokes the "prefire" and "getData" methods of processes. These methods would apply the initial properties variable data on processes and update the "runtime properties" with new values coming from input port or initial properties. Following that, the thread would invoke the "fire" method of the process to perform its main functionality. The thread then updates runtime properties of the process, whilst process is able to fire. The ability to fire is identified in the "canFire" method, which it delegates this task to the "isFireable" method of the process). The "isFireable" method checks the availability of resources and identifies whether process is able to run or not. The "canFire" and "isFireable" methods are dependent on the strategy of process thread. PNA performs the following sequence of methods (the asterisk denotes methods that may be executed zero or several times during each execution).

init, prefire,getData,(canFire,fire*,getData)*,postfire

```
public void run()
	{
		strategy.preFire(process);
		Map data = process.getData();
		while(strategy.canFire(process,data))
		{
			strategy.fire(process,data);
			data = process.getData();
		}
		strategy.postFire(process);
	}
```

Figure 3.8. The run method of “Process Thread”.

3.4.1 A simple Producer and Consumer process network in PNA

The producer consumer process network, showed in Figure 3.6, contains a producer process that generates a series of tokens. The producer process has one output port that is connected to channel that is implemented by half-channel. The produced tokens from output port of producer process are sent to input port of consumer process via the channel. The consumer process just prints out received tokens.

The main class of this producer and consumer process network is shown in Figure 3.9. This case study uses “PNAKahnFactory” which creates entire components in PNA. Whenever the process is created by the factory, a scheduling thread is assigned to the process to identify a MoC and scheduling policy in the process. The dataflow scheduling strategy is considered for the producer and consumer processes in this case study. The processes are initialized by setting the process to network, and determined the number of input and output ports. The range of generated tokens in producer process, which is from 1 to 5, is specified as a property of process by “updateProperties”. The default initial and upper bound of generated tokens in producer process are zero, which is identified in constructor of “TokenProducer” class. The consumer process does not require any initial properties. Processes are connected together by connecting the output port of producer process to input port of consumer process. The connections construct PN and it is ready to run. This method starts execution of PN, by adding all the channels and processes to network and triggering process’ threads.

```
public class PNASimpletest {
      public static void main(String[] args) {
        /*
         * decide which factory do you need?
         * 1- PNAKahnFactory: running kahn PN, standalone
         * 2- PNAMOPFactory : running kahn PN and Enigma MOP
         * 3- PNACompFactory: running kahn PN and Enigma MOP with meta-computation
         */
         Factory factory = PNAKahnFactory.newInstance();
        //Factory factory = PNAMOPFactory.newInstance();
        //Factory factory = PNACompFactory.newInstance();

        // create a network
        Network network = factory.newNetwork(factory);
        Builder builder = factory.newBuilder(network);

        // create the processes
        // Integer Generator
        Process producer = factory.newProcess(network,"au.edu.adelaide.pna.processes.
        TokenProducer");
        HashMap producerProperties = new HashMap();
        producerProperties.put("Initial Value", "1");
        producerProperties.put("Upper Bound", "5");
        producer.updateProperties(producerProperties);
        builder.add(producer);

        // print the network result
        Process consumer =
        factory.newProcess(network,"au.edu.adelaide.pna.processes.TokenPrinter");
        builder.add(consumer);

        // connect the producer and consumder - this handles in the builder
        builder.connect(producer, producer.getOutputPort(0), consumer,
                        consumer.getInputPort(0));
        // start computation
        builder.trigger();
        try {
                Thread.sleep(1100);
                System.out.println("Exiting");
                System.exit(0);
        } catch (Exception e) {
        }
      }
  }
```

Figure 3.9. PNA main class for producer and Consumer process network.

3.4.2 Implementing a Process Network case study

The experiment shown in Figure 3.4 is implemented in PNA. The case study in Figure 3.4 is constructed in “PNATest” class that is shown in Figure 3.10. In this class which is a Main class, mentioned processes in Figure 3.4 including two “TokenProducer”, two “Add”, “TokenSplitter”, “Multiply”, “Minus” and “TokenPrinter” processes. The required information of each process such as initial value, upper value or operation is added into a data structure, which is assigned to the process. Processes are connected together in form of connecting output port of producer to input port of consumer. This connection is a

representative of channel in PN. The case study implemented in this section is used in subsequent chapters, particularly in chapter 7.

```
public static void main(String[] args) {
        //Start Specification
        Factory factory = PNAKahnFactory.newInstance();

        // create a network
        Network network = factory.newNetwork(factory);
        Builder builder = factory.newBuilder(network);

        // create the processes
        /  Integer Generator
        Process producer1 =
        factory.newProcess(network,"au.edu.adelaide.pna.processes.TokenProducer");
        // adding description to Token producer process
        HashMap producerProperties = new HashMap();
        producerProperties.put("Initial Value", "1");
        producerProperties.put("Upper Bound", "5");
        producer.updateProperties(producerProperties);
        builder.add(producer1);

        //Add process
        Process add
        =factory.newProcess(network,"au.edu.adelaide.pna.processes.ArithmaticProcess");
        // adding description to add process
        HashMap addProperties = new HashMap();
        addProperties.put("operator","+");
        add.updateProperties(addProperties);
        builder.add(add);

        //Copy one input into to two outputs

        Process splitter =
        factory.newProcess(network,"au.edu.adelaide.pna.processes.TokenSplitterProcesss");
        builder.add(splitter);

//                      Creating and adding the rest of processes

        // connect the producer and consumer
        builder.connect(producer, producer.getOutputPort(0),add, add.getInputPort(0));
        builder.connect(producer1, producer1.getOutputPort(0), add, add.getInputPort(1));
        builder.connect(add, add.getOutputPort(0), splitter, splitter.getInputPort(0));

//                      Connecting the rest of processes together
        //end Specification

        // start computation and execution
        builder.trigger();
}
```

Figure 3.10. PNA main class for process network case study.

3.5 Summary

In this chapter, PN in terms of its MoC is studied in the context of Kepler workflow systems to present adequate representative of scientific workflow systems. We express the notion of scientific workflow system in PNA that its design and implementation are explored in this

chapter. We will use this notion of PNA in subsequent chapters particularly on chapter 7 to implement adaptive provenance collection mechanisms on it.

We explore the concept of MoC and how interpreter applies it on dataflow. The Kepler workflow system combines the MoC and interpreter in a director that has wide variation to cater for scientist requirements of conveniently using and designing scientific workflow systems. PN is one of the Kepler directors that appropriately represent scientific workflow systems.

In the next chapter, we describe the model of provenance used in the design and development of our provenance collection mechanisms. It determines which sort of provenance information should be collected, the level of detail at which it should be collected and how that information is represented.

4 MODEL OF PROVENANCE

A *Model of Provenance* (MoP) (not to be confused with MetaObject Protocol (MOP)) determines what provenance information should be collected and how that information should be represented. There are various design choices for provenance mechanisms in workflow systems, which results in the existence of a variety of MoP. This chapter aims to make explicit the concept of MoP, survey different models in related literature and propose a MoP for case studies presented later in this thesis.

In this chapter, we firstly link the concept of MoP to Model of Computation (MoC) and interpreters in workflow systems. Following that, in section 4.2, a simple MoP is elaborated. In section 4.3, we provide a survey and summary of various MoPs, in other works, that could be applied to scientific workflow systems. The Anand [142], COMAD [142, 152] and Muniswamy-Reddy [72] MoP are introduced to illustrate various approaches modelling provenance information and dependencies.

In section 4.4, the general Open Provenance Model (OPM) [49, 153], and Open Provenance Model for Workflows (OPMW) [52] MoP are presented. These are concerned with enabling interoperability between provenance systems that store and query shared provenance information differently; or have a different representation for the same provenance information [53, 154]. The OPM MoP is examined in terms of its interoperability across provenance systems.

The concept of MoP is linked to provenance granularity in section 4.5. We explore the definition of coarse, medium and fine-grained provenance to clarify this connection. In this section, a MoP is elaborated to represent each level of provenance granularity. Also, some applications of designed coarse, medium and fine-grained MoP are presented.

4.1 Introduction

Provenance in scientific experiments aims to provide an understanding of final results by examining the sequence of steps leading to these results. It investigates what was happened in the system during a specific process in the past by providing a chain of reasoning. More

broadly, provenance is intended to answer many questions, as listed in section 2.1.1. However, in this work, we are interested in answering questions regarding sequence and history of data products [21, 44-46] in the context of scientific workflow systems although many other valid questions could be answered under the broad definition of provenance.

The concept of provenance (also known as Lineage) has a direct relation to the concept of dependency. In common usage, the two terms: Lineage and Provenance, are almost synonymous, but in the context of scientific workflow systems several authors [45, 155] refer to lineage as the dependency chain associated with deriving a given data product. Therefore, lineage of derived data product describes the dependency chain of processes (transformations) that are applied to derive the data. Provenance would be associated not only with data products, but also with the processes of data creation [45]. Provenance information addresses information about dependency within or between data and processes [21] (in other words, the relationship between produced and consumed data; the relationship between processes; and the relationship between data and process) .

Our focus here is on provenance as used in workflow systems, but provenance more broadly is used in many areas and applications, such as database, data quality, audit trail, replication recipes, informational data discovery [45]. As explained in section 2.1.1, there are various general views on provenance such as workflow and data provenance view [75], or view of provenance system treating the workflow components as black-, white- and grey-box [20, 24, 76, 77]. In some views [20, 24, 75-77], processes in workflow systems are treated as black boxes and its provenance is referred as coarse-grained provenance. However, the data provenance is referred as fine-grained provenance, representing the sequence of creation of data in a system's components is captured at the database level.

Recall in section 2.1.1, provenance systems, in the context of scientific workflows involve a (dynamic) mechanism that collects, presents and manages sufficient levels of static and dynamic workflow information for different purposes such as reproducing, inspecting and validating experiments' results. In this section, we briefly elaborate the concept of provenance. In the next section, a MoP is explored in detail and illustrated with a simple case study.

4.2 A Generalized Notion of Model of Provenance

In this section, we explore the concept of MoP through its connection to a MoC and its corresponding interpreter. We also illustrate this connection in a simple case study.

As explained in section 3.1, scientific workflow systems can be explained and executed under different MoCs, corresponding to the underlying operation of scientific workflow and also defining how execution semantics are applied to the dataflow. The semantics of a MoC in a dataflow computation is a set of rules governing the interactions of dataflow components that can be expressed with an interpreter. An interpreter directly encodes and applies rules of MoC on a dataflow (or in more general terms the workflow system). The interactions of dataflow components expressed by a set of rules, reveal the dependencies either within or between components. These dependencies can be collected as a form of provenance information and represented in a MoP. Therefore, provenance has a direct relationship to the rules in MoC and the way interpreter encodes the rules on a dataflow.

For example, existence of data on the input port of a process activates the process and makes it ready to run (i.e it enters a fireable state). This is one possible execution semantic that can be defined in rules (of MoC). Sending produced data immediately to subsequent processes is another rule. Using these two rules, the input data (I1) in Figure 4.1 activates a process (P1), following that computational steps occur on the input data to transform it to output data (O1). The output data (O1) is transferred to subsequent process (P2). Therefore, as data produced by first process (P1) is sent to the successor process (P2), and P2 is made fireable. In this case, execution of the successor process (P2) is dependent on first process (P1) and this dependency is established by the edge from P1 to P2. The set of firing rules in the MoC establish as a set of provenance dependencies that need a MoP. The MoP defines a set of rules to represent provenance information and dependencies based on the firing rules in the MoC.

Data defines the interactions between scientific workflow (dataflow) components. A component interacts with the subsequent components by transferring its produced data to that component. This is what a provenance system records as the sequence of creation of data. An interpreter encodes rules in a MoC, which results in establishing those interactions between workflow components. The process of provenance collection can collaborate with interpreter

and rules in the MoC (such as the COMAD director [142, 152] in Kepler explained in section 4.3.2). Thus, provenance collection mechanisms can be superimposed onto the rules for dataflow semantics, as shown in the previous paragraph.

In our view, ideally a MoP will specify provenance information and representation model of the information. However, this is not what always happens in practice. In practice, there are different systems for collecting provenance operating on workflows, which make it hard to characterize a generic set of rules for both collection and representation of provenance in all provenance systems.

There is a variety of MoP, ranging from a simple execution trace to a fairly complex and detailed MoP. We present some of them in section 4.3. In the next section, a simple MoP is presented to demonstrate basic concepts.

4.2.1 A Simple MoP

In this section, a simple MoP is introduced to represent provenance of the dataflow graph presented in Figure 4.1 (it is similar to dataflow graph to Figure 3.4). In this section, we explore a simple MoP to provide an understanding of how a simple provenance collection mechanism can work. We show how to express provenance collection by defining a single MoP on top of the single MoC defined in previous chapter. We focus on collection of provenance information based on our simple MoP and representation will be discussed later. We have explained a set of firing rules expressing MoC for a simple dataflow case study in section 3.1.1. We are going to elaborate a set of provenance rules expressing a MoP according to the firing rules expressed in MoC of this simple dataflow case study. This MoP introduces dependency patterns (including data-process-dependency, data-dependency and process-dependency, explained in following paragraphs). A simple MoP is considered as a history and record of the input and output of each process invocation during a workflow run. To see this, consider Figure 4.1 that shows inputs (I1) and its output (O1) of first node (P1) that goes into second node (P2's) input. It also shows P2's outputs (O2) and inputs (I2). In this case, P2's output (O2) directly depends on its inputs (I2) while O2 indirectly depends on P1's inputs (I1).

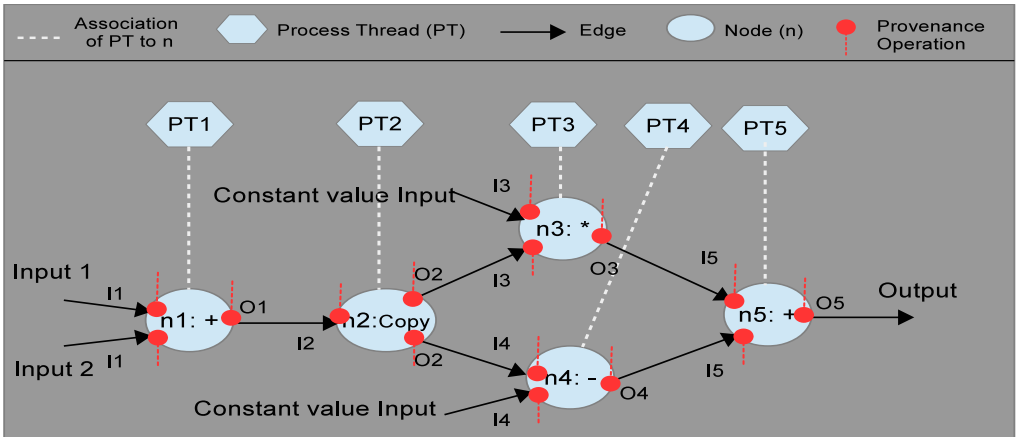


Figure 4.1. A dataflow graph

Our simple MoP is expressed by a set of rules regarding how provenance (particularly provenance dependency) is collected and then represented. A provenance system has a provenance collection mechanism with a number of provenance operations. The provenance operations are added (hooked into) to the dataflow graph aiming to collect specific sort of data required for constructing provenance information (as explained below). In the simple MoP, as shown in Figure 4.1, provenance operations are added to the input ports and output ports of processes (this is analogous to saying that they are added on input and output edges because edges are connected immediately to ports; and ports do not manipulate data). There is no provenance operation attached to the computational steps inside processes because the intra-process activities in processes are ignored (for now, we consider processes as black boxes). The added provenance operations on input and output ports of processes collect input and output data that is then sent to the provenance collection mechanism in order to construct and represent provenance information.

The described simple MoP derives a number of dependencies that are shown in Figures 4.2, 4.3 and 4.4. In the Figures, data (called token or artifact) is represented by tokens whose labels start with "A"; process or actors (that it has one or a number of tasks) are represented by labels starting with "P". Processes are connected together by edges. These elements present constituent components of a dataflow graph; however, they need an entity to manage and control their collaboration and execution. This controlling entity in dataflow graph and workflow system (such as interpreter, agent or director) directs processes and supervises the execution of processes as explained in section 3.1. Figure 4.1 and Figure 4.2 represents this controlling entity is represented by Ag.

As mentioned earlier, provenance operations are added into input or output ports of processes to construct provenance information. As a data token enters a process, provenance operations operating on input ports of processes are notified and collect input data. The operations inform the provenance collection mechanism regarding input values. The provenance collection mechanism constructs and represents provenance information for this event as it receives the input value from the provenance operations. The provenance information includes,

- *Artifact*: the input value is added into a list of provenance data (artifacts) (if it were not in the list).
- *Process*: the process that the operation is attached into is added into a list of processes (if it were not in the list).
- *Dependency*: a data-process dependency between processes and input data is constructed, for example data-process dependency between P2 to A3 as shown in Figure 4.2.

The same scenario occurs when the output data is put into output port (or edge), which provenance operation, working on output ports informs the provenance collection mechanism. The collection mechanism constructs provenance information similarly in a form of adding data and processes into the list of data and process; and constructing a data-process dependency.

For example, the provenance operation working on the output port of P1 captures the output data (O1). The provenance operation informs the provenance collection mechanism of this action on the output port, after which, the provenance collection mechanism, firstly, will check the existence of the O1 into the list of artifact and will add O1 into the list (if it was not in the list). In the next step, the provenance collection mechanism also checks the existence of P1 in the list of process in order to add it into the list if it was not in. Noting that it would not be added into the list because it has already been added. Finally, as shown in Figure 4.2, a data-process dependency between A4 to P2 is constructed that denotes the flow of data through processes during workflow run. In the Figure 4.2, there is an agent for each process, because the dataflow graph is a PN that each process has it associated process thread. The process thread is considered as an agent because it controls and manages the execution of a process.

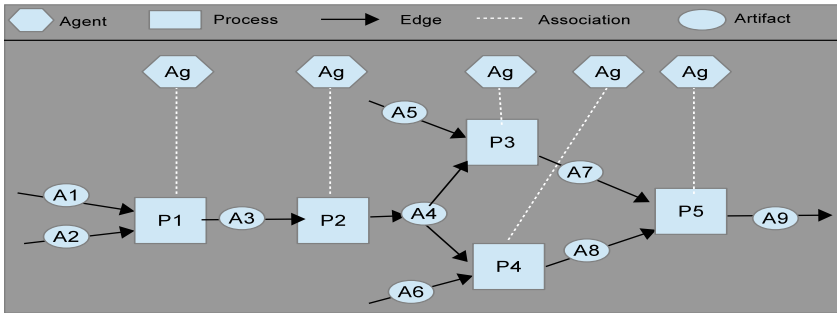


Figure 4.2. Data-process dependency.

The provenance collection mechanism constructs two other types of dependencies, including data-dependency and process-dependency, as shown in Figure 4.3 and 4.4, respectively. A data-dependency shows the relationship of derivation of output data from input data without considering the process causing this transformation, for example, a data-dependency exists between A4 to A3 as shown in Figure 4.3. When a provenance operation reports that an output data is put into an output port (via an edge), the provenance collection mechanism constructs and represents this dependency according to the input and output values received from provenance operations working on input and output ports.

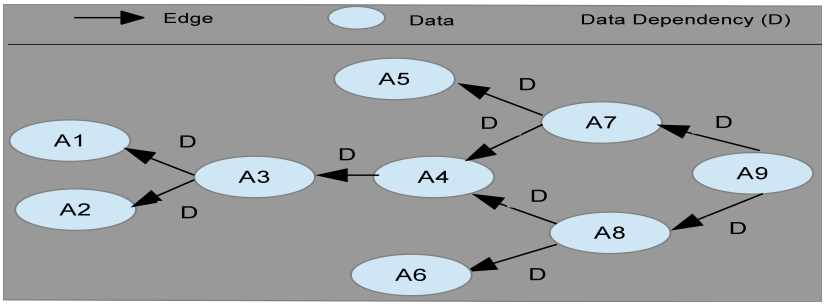


Figure 4.3. Data dependency.

Following the dataflow rules, a process will put the output result (data) on an output edge to activate the subsequent process connected to the edge. The transferred data will then trigger the subsequent process. Therefore, there is dependency between the processes called process-dependency. When a provenance operation working on input ports of a process reports the entrance of a data value into a process, the provenance collection mechanism constructs a process dependency between the processes (consumer process) that the operation is attached to its input port, and a predecessor process (producer process) as depicted in Figure 4.4. The process dependency (in Figure 4.4) is also used in business workflows in order to capture the sequence of processes' execution and validate the logic of the workflow's tasks.

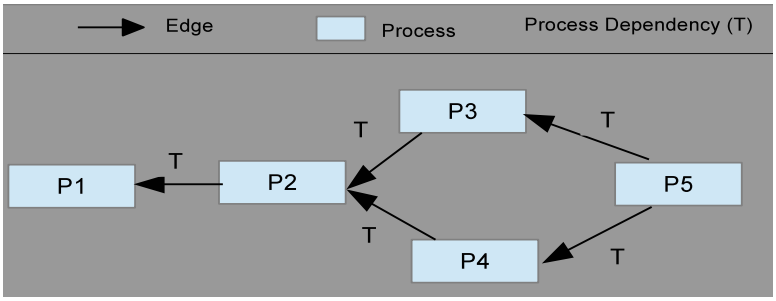


Figure 4.4. Process dependency graph.

A set of rules for provenance have been represented for our simple MoP early in this section (see above, immediately after Figure 4.1). The set of provenance rules is constructed according to a set of rules representing MoC for the dataflow case study (described in section 3.1.1). Our simple MoP is useful to represent provenance dependencies in a dataflow with following assumptions.

- The intra-process activities in processes are ignored. Processes have a single computational step (single firing) and considered a black-box.
- The output is determined solely by the current inputs.
- All inputs in a process need to be used to produce outputs (all outputs depend on all inputs in an invocation).

These assumptions mean that this MoP represents provenance information in a graph with atomic data structure and atomic (single-firing) process nodes. Note that, many scientific workflows don't follow these assumptions (as shown in section 4.3), because they might use

- Processes that will be wrapped complex tasks (such as, external applications and services).
- Composite processes (a process containing a number of other processes).
- Process with multiple tasks invocations (such as Kepler multiple-firing actor model, explained in section 3.2.1).
- Non-atomic data structure such as composite data (nested data or complex structure of data); or stream data structures (where different parts of the structure are generated at different times, which can lead to more complex semantics). There are some issues of copying and updating a sub-section of the non-atomic data structure.

As we will explore in section 4.3, the non-atomic data structure leads to various dependency patterns. For example, the dependency between output and input data could be the dependency of outputs on a single input or a subset of inputs instead of dependency on all inputs. Process with multiple task invocations is another example that leads to variety of

dependency patterns. In this model, tasks inside a process might be invoked a number of times by interpreter. Therefore, in this situation our simple MoP infers neither sufficient nor correct data dependencies, because processes are not atomic.

In summary, we have introduced a simple MoP with certain assumptions (atomic data structure and single-firing process nodes) but most workflow systems do not following these assumptions, as explained in previous paragraph. Our simple MoP provides an understanding of how a simple provenance collection mechanism can work. We will propose a MoP in the section 4.5, which is used in our provenance collection mechanism (designed in chapter 6 and implemented in chapter 7). In the next section, we are reviewing and summarizing quite separate MoPs proposed in variety of literatures.

4.3 Reviewing Model of Provenance

We explored the concept of MoP in section 2.1 and early sections of this chapter. Now, in this section, we review and summarize a number of separate MoPs from the literature. We aim to investigate how these MoPs work and how they tackle different aspects of Model of Provenance (MoP). This investigation provides a sufficient level of understanding of how to identify a set of underlying principles and design a conceptual framework. The conceptual framework would abstract the set of underlying principles in a provenance model.

The reviewed MoPs in this section differ in various ways from our simplified view of MoP outlined in the previous section. In particularly, most of them do not explicitly present an underlying MoC for their work (except COMAD which is explained in section 4.3.2). We describe their view in terms of their description about provenance. We present an understanding of provenance and MoP in this section and we will return to the view in section 4.4 where we discuss the generalized Open Provenance Model. A number of MoPs including Muniswamy-Reddy [72], Anand [142] (they named by the paper's first authors name), COMAD [142, 152] are reviewed to provide a fairly broad view of this concept.

4.3.1 Anand's MoP

Anand [142] takes the view that many of MoPs are similar to our assumptions for a simple MoP in terms of using atomic data and atomic (single-firing) process nodes, explained in section 4.2.1. Therefore, capturing the data dependencies in these models is simple. In contrast, capturing data dependencies becomes complicated when workflows need to handle non-atomic data structures (refer to section 4.2.1); support multiple-invocation in processes (refer to section 4.2.1); and/or capture fine-grained data dependency. In such a more difficult model, data relationships are expressed in three dimensions:

- *Flow relation* is a flow of input to output data
- *Parent-child* relation is a structural relationship among nodes (contains either processes or tasks). It includes the relation among tasks or process of structures, particularly in case of composite processes.
- *Lineage relation* is a relation among nodes and invocations. It represents the data and process dependency for the nested data structures indicating the fine-grained dependency [142].

We explained a number of dependencies, including data-process-dependency, data-dependency and process-dependency, presented in a simple MoP in section 4.2.1 for simple scientific workflow systems (having simple process-dependency over atomic data (artifact or token) and atomic processes). However, a number of other dependencies have arisen in a case of having workflow systems with different structures, including

- *Multiple-task invocations* dependency between processes (additional dependencies between separate invocations that are presented in following Multiple Invocation Model)
- *Non-atomic data model* such as nested data presented in XML presented in following Nested Model.
- *Fine-grained dependency* (known as token-level dependency), relating nodes (contains either processes or tasks) based on dependency of tokens, rather than using coarse-grained structure (such as, considering input and output edge of flow graph) that is presented in following Unified Model. Fine-grained dependency facilitates lineage relations presenting the invocations that lead to create a node bases on its descendent node [142].

Anand [142] proposed a number of MoPs as explained below, ranging from a basic model to a complex one such that each model represents provenance information using one or several above dependencies.

- *Basic Model* represents provenance information in a workflow system, including process-dependency over atomic artifacts and (single method invocation in) atomic processes with stateless process (processes' invocations without dependency to other invocations). The defined MoP in basic models is similar to our simple MoP defined in section 4.2.
- *Multiple Invocation Model* represents provenance information in a workflow system with multiple task invocations in processes over streams of data (performing a number of invocations (firing) over a stream of input data). For example, suppose that firing rules (in MoC) of Process Networks are employed in a dataflow system. If tasks are stateless (no information and memory needs to be preserved between invocations), the invocations are independent, and concurrent execution is performed over the input stream. While, in case of dataflow with stateful tasks (preserving information between invocations), invocations are dependent on the input stream of previous invocations; causing additional dependencies between separate invocations (which is not covered in the core OPM and simple MoP).
- *Nested Model (Copy and Update Semantics)* represents provenance information in a workflow system using non-atomic data structure such as nested data (presented in XML). This model deals with the issue of copying and updating a sub-section of complex structure of data. Copy semantic model maps input data structure to an XML tree data structure [142]. An update semantics model considers invocation as a set of updates. In the case that invocations are independent, then these updates produce new version of their inputs instead of producing all the inputs for every invocation, in order to have efficient workflow execution and provenance storage [142].
- *Unified Model* represents provenance information in a workflow system using non-atomic data model with update semantics and multiple-invocation processes. The Unified Model also represents another type of data dependency, which is called Fine-grained data dependency[1] [142] (and is also known as token-level dependency). Fine-grained data dependency is a dependency among nodes (contains either processes or tasks) and invocations, representing the data and process dependencies for the nested data structures. It is presented by lineage relations presenting the invocations that lead to activation of a node by its descendent (caller) node [142]. Finally, the Unified Model can capture all Flow, Parent-child, and Lineage relations.

In summary, a number of workflow structures, which result in various forms of dependencies have been explored in this section. Following that a number of models that address the dependencies have been introduced.

[1] This is different from fine-grained provenance.

4.3.2 COMAD MoP

Collection-oriented modelling and design (COMAD) [142, 152] is a director of Kepler scientific workflow system. The view of understanding and modelling provenance in COMAD is explicitly presented with an underlying MoC (because COMAD is a director). The COMAD director is a supervisory program that knows how to deal with input data, how to process input data and produce output data and in addition knows how workflow components interact. The COMAD has access to all the rules of MoC establishing interactions between workflow components and the process of producing and consuming data. Therefore, the COMAD director can capture provenance information in a workflow systems using complex structure and nested data collection. These structures in workflow systems cause the creation of several dependencies that have a fairly low level of details and most provenance collection mechanisms cannot collect them.

The COMAD MoP addresses several dependencies (including all dependencies introduced in Anand MoP [142]) arisen by fine-grained dependencies, multiple task invocations of processes, and nested data collection. It also implemented and examined in the Kepler scientific workflow system.

4.3.3 Muniswamy-Reddy's MoP

Muniswamy-Reddy [72] puts forward an Abstract Provenance Model containing the Ideal Model, Information Flow Model, Control Flow Model and Causality Model that are represented (instantiated) by either Provenance-Aware Systems (PASS) (explained in chapter 5) or Open Provenance Model (OPM) model (explained in section 4.4). The Observed, Disclosed and Complementary [72] provenance collection mechanisms (explained in section 2.1.3) are proposed for these models. In this regard, four properties are defined in [56, 72] as desirable attributes of provenance systems that are satisfied by their design methodology in PASS. These properties indicate when a data item and its provenance are recorded in provenance systems. The properties (called data requirements in [72]) include Provenance Data-Coupling, Multi-object Causal Ordering, Data-Independent Persistence and Efficient query state that are described in following.

- *Provenance Data-Coupling:* provenance should accurately describe recorded data (called Provenance Data-Coupling). The Provenance Data-Coupling enables users making accurate decision about using provenance. For example, without this property, users might use old data with new provenance (or vice versa) which causes the users working on the provenance to be misled into using invalid data [56, 72].
- *Multi-object Causal Ordering:* provenance should have the chain of ancestors' relationship among objects (recorded data items). Therefore, the ancestor object (and its provenance) must be recorded and stored before recording and storing the object in order to preserve the causal ordering (called Multi-object Causal Ordering). The Multi-object Causal Ordering prevents dangling pointer in the provenance in case of occurrence of a system crash before storing the ancestor object of a stored object. A dangling pointer in the provenance is created, if an object and its provenance are stored and system crashed before storing it ancestor object and its provenance [56, 72].
- *Data-Independent Persistence*: provenance information of the ancestor should be retainable even if the ancestor data is removed. Therefore, then if P is ancestor of Q, the provenance information of Q would contain the provenance information of P, even if P is removed and is not accessible anymore. The provenance of P would only be deleted when P had no descendent anymore (called Data-Independent Persistence). Data-Independent Persistence ensures that the provenance are accessible even in case of objects deletion [56, 72].
- *Efficient query state*: provenance systems should support efficient queries on provenance data (called Efficient query state) because users want to access and verify provenance properties in their data [56, 72].

Muniswamy-Reddy's Abstract Provenance Models includes the following MoPs,

- *The Information Flow Model,* which is a MoP that represents flow of information (data) between objects. In this model, a source object is the ancestor of the destination object, which "ancestor" label is annotated on edge between objects [72]. In this scheme an object could be data, a file, a process or an entity of a workflow system such as a workflow engine.
- *The Control Flow Model*, which is a superset of Information Flow Model. It is structurally similar to the Information Flow Model and it represents control dependencies (by annotating node and edge with appropriate attributes) as well as flow (ancestor) dependency [72].
- *The Causality Model* that is a MoP represents a set of events that has happened and also possibly will happen in future to influence the output results. In this model all causal relationships and dependencies are defined and will effect of the model of execution and results. For example, in this model, all the input data would influence output data, even if input data were not read and involved [72].

- *The Ideal Model* is a MoP that represents sufficient information about workflow entities to recreate them exactly, and capture exact dependencies. The ideal model would be possible to be used only when we have a deterministic system [72]. It is a comprehensive MoP that captures everything at very detailed level to be able to emulate the workflow specification and execution.

Muniswamy-Reddy [72] proposed PASS and OPM as two representation models that determine how provenance information should be represented. He explains PASS and OPM in the context of defined concepts in abstract provenance models. PASS is defined based on an Information Flow Model (that captures limited control flow information). The causality model can be expressed by OPM, although OPM has an annotation style similar to the Control Flow Model. OPM is introduced to make provenance interoperable [53, 72].

In preceding sub-sections, we have summarized quite separate MoPs. The Anand, COMAD and Muniswamy-Reddy MoPs represent varieties of dependencies and structures of provenance information. The MoPs presented in the next section are concerned with providing interoperability in provenance systems. Interoperability in provenance systems is the ability to use provenance information exchanged from other systems.

Up to now, we have presented the varieties of MoPs that have some diversity in their modelling structure. This diversity of provenance model and the importance of collaborations employing workflow systems generating provenance information from different sources emphasize the need for interoperability. Interoperability in provenance systems across platforms (including workflow system and heterogeneous applications in collaborative works) enables participants to store and query shared provenance information in their way (which might be different from other ways); and/or to have different representation of the same provenance information [53, 154]. Ellqvist [156] proposed a mediator architecture called Scientific Workflow Provenance Data Model (SWPDM). The mediator integrates provenance information (with others models of provenance) from multiple sources and finally makes them interoperable together but not in a uniform and widely acceptable view [156].

It is clearly desirable to have common view of provenance. Therefore a consensus has emerged among different groups for a common view towards a MoP in workflow systems. The result was a generic MoP, called Open Provenance Model (OPM) [49, 153], to address

the challenge of interoperability in provenance systems. In the following section, we will discuss how OPM addresses issues of interoperability.

4.4 Open Provenance Model MoP

OPM is a widely accepted and domain-independent MoP, which is the result of a significant effort on standardization and exchange in the scientific workflow system through a series of provenance challenges. The first Provenance Challenge [94], with the goal of understanding different provenance systems' capabilities and the expressiveness of provenance representations, are followed by the second Provenance Challenge intending to establish an interoperable provenance system. The first version of Open Provenance Model (OPM) was released as a discussion during a workshop in Salt Lake City [94]. Version 1.01 [153] with the aim of identifying the OPM specification in 2008 was followed by the version 1.1 [49] aiming for exchangeable provenance information and answering precise provenance queries (known as the third Provenance Challenge), which resulted in an open-source model for the governance of OPM [49, 53]. Eventually, Fourth and Last Provenance Challenge [50] aims to apply OPM to a broad end-to-end scenario, it also presents novel functionality in a favor of interoperability in provenance system.

Many scientific workflow systems support OPM in their provenance model, such as Kepler, Tavarna, SPADE, Karma, Swift, PASS, VisTrails and Trident (we will discuss these in chapter 5). Every OPM graph is presented in a form of a directed acyclic graph (DAG) [22]. OPM's DAG consists of nodes (process, artifact and agents) and edges. An edge represents causal dependency between nodes: an arrow starts from effect and ends in cause [49, 153]. In the model presented in Figure 4.5, a node can be an artefact, such as an immutable piece of state (considered as a data); process as transformational (computational) action producing new artifacts (considered as a task, actor or an actor's invocation); or an agent as an individual control process that direct processes. Edges in OPM can express one of the following causal dependencies, as also shown in Figure 4.5.

- *Used (R)* reveals the causal dependency of a process to one or a number of artifacts, indicating completion of a process execution was required to use of artifact(s) [49]. For example, as shown in Figure 4.6, the completion of execution of the process P1 is required to use (and consume) A1 and A2. Therefore, P1 used A1, and also P1 used A2.

- *WasGeneratedBy(R)* reveals the causal dependency of one or a number of artifacts to a process, indicating a process execution was required to be initiated for the artifact(s) to have been generated [49]. For example, as shown in Figure 4.6, the execution of process P1 needs to be initiated to generate A3. Therefore, A3 was generated by P1.
- *WasControlledBy(R)* reveals the causal dependency between a process and an agent, indicating the start and end of process (P1) was controlled by agent (Ag1) [49]. For example, in Figure 4.6, Ag1 was controlled by P1.
- *WasTriggeredBy()* reveals the causal dependency of a process (P2 shown in Figure 4.6) to a process (P1 shown in Figure 4.6), indicating the starting a process (P2) was triggered by the completion of a (predecessor) process (P1) [49].
- *WasDerivedFrom()* reveals the causal dependency of an artifact (A2 shown in Figure 4.6) to another artifact (A1 shown in Figure 4.6), indicating an artifact (A1) is required to have been generated for artifact (A2) to be generated. That is the artifact (A2) generating process was derived from presence of (predecessor) artifact (A1) [49].

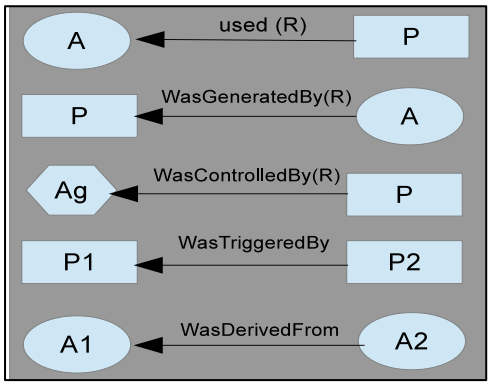


Figure 4.5. OPM dependencies [49].

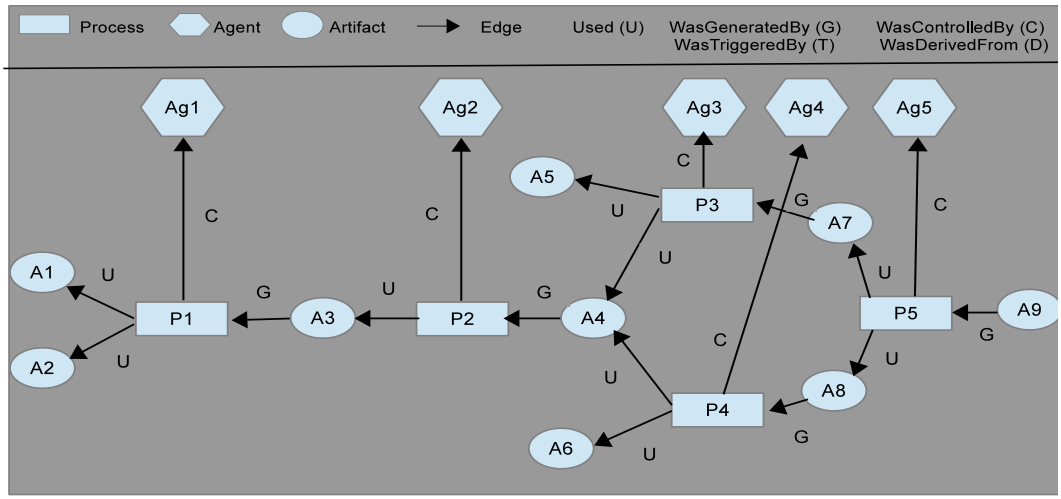


Figure 4.6. OPM dependency graph of Figure 4.1.

Figure 4.6 describes the OPM model of the data process graph shown in Figure 4.1. Figure 4.6 shows the causal dependencies between artifacts, processes and agents, such as "WasControlledBy", "WasGeneratedBy" and "Used". The "WasControlledBy" dependency is a dependency between a process and an agent (that directs process execution). In Figure 4.6, the dependencies of artifacts and processes are presented in processes that use or generate artifacts. The "WasDerivedFrom" and "WasTriggeredBy" dependencies are not shown in this Figure to avoid confusion, they are similar to data-dependency and process-dependency shown in Figure 4.3 and 4.4 respectively.

We now look at the fundamentals of OPM to elaborate its application in practice. As an example, we will look at the well-known Kepler workflow system (explained in section 5.2.1). OPM entities and relationships are expressible in the Kepler workflow system in a way that, artifact and process correspond to the concept of data token and actor, respectively; thus, a process usually uses one or more tokens and generates one or more new tokens. An OPM agent is considered similar to the concept of director in the context of Kepler workflow system; however, Kepler's director needs to handle some other tasks, for example to control an execution such that it proceeds in a manner consistent with the semantics of the underlying model of execution. All these maps can be made explicit in an OPM Profile (defined in the next paragraph) for Kepler workflow provenance system.

Now, we discuss the important concepts of OPM including Profile, Role, Account and Annotation. An OPM *Profile* [22, 49, 52, 157] aims to define a specialization of OPM in the form of an OPM graph, in order to formalize the best practice (for handling a specific problem) and guidelines for a specific domain or to handle a specific problem [22, 49]. In the third provenance challenge, it was agreed that extensibility is one of the most important characteristics of OPM, which needs to be captured by a notation for profile [50], such as a profile representing the notion of "collection" [50] in order to model a set of artifacts in a collection; and Open Provenance Model for Workflows (refer to section 4.4.1) [52] in OPM . In [157], Profile is used to show how the Nested Relational Calculus (NRC) (complex) data model for a workflow repository could be mapped to OPM. Ooms [22] proposed an OPM profile that maps workflow execution and OPM entities. This profile maps data to artifacts, workflow tasks to processes and actors services to agents, and expresses how the OPM

relations are constructed during workflow execution. We will present a profile mapping workflow events to OPM entities for our MoP in section 4.5.1.3.

A *Role* [49] assigns a type of activity to identify how artifacts, processes or agents interact with one another (it usually indicates an artifacts' or agents' function in a process). It is indicated by the tag "R" in the dependencies (shown in OPM causal relationships in Figure 4.5) to distinguish the dependency nature when we have multiple edges connecting to a process. A role may be used in different situations such as, when a process uses more than one artifact, when an artifact is used in more than one process, when an agent controls more than one process, or when a process is controlled by more than one agent. For interoperability purposes a set of standard roles with agreed meaning in communities needs to be defined by means of a Profile [49].

An *Account* [22, 49, 75, 158, 159] provides a view of provenance information or subset of provenance information [49, 158] representing a description at some level of detail that is provided by one or more observers. We use accounts during a collection mechanism to specify views of provenance information at various levels of granularity. OPM entities and dependencies can have multiple accounts. Accounts present a view of provenance information, for example, fine-grained and coarse-grained. Fine-grained and coarse-grained provenance have some artifacts and processes in common. The accounts can participate in a *Refinement* relation. An account (Ac1) is a refinement of another account (Ac2) if the OPM graph represented by Ac1 is a more complete description of OPM graph represented by Ac2 [159]. A fine-grained account is a refinement of coarse-grained account, because all data derived from the coarse account are also held in the fine account. An account (Ac1) is a refinement of another account (Ac2) if the set of dependencies inferred in Ac1 is a superset of Ac2 dependencies after the application of completion rules [49] (we will describe the completion rules or inference rules in section 4.5.2). We will define three accounts referring to fine-grained, medium-grained and coarse-grained provenance views in section 4.5.

An *Annotation* [49, 158] allows the OPM model to represent extra information. The extra information is represented as a form of annotation that enables meaningful exchange of provenance information for interoperability purposes [49].

OPM intends to standardize and provide a widely acceptable MoP, while there are still a number of dependency patterns and design issues [142] which may be not supported by the core OPM model (some of them are explained in section 4.4.2). Therefore an OPM profile makes it possible to express some of those situations and eventually makes OPM an extensible model to some extent. Open Provenance Model for Workflows [52] is one of the OPM profiles that is designed for this purpose and is explained in the next section.

4.4.1 The Open Provenance Model for Workflows MoP

The Open Provenance Model for Workflows (OPMW) [52] is an OWL ontology representing abstract workflows in addition to workflow execution traces, in order to fit into scientific workflow domains. OPMW mainly focuses on making scientific workflow provenance available in the form of Linked Data [52, 158], which helps in mapping OPMW to another models (such as PROV [51]) and producing an RDF [52, 158]. Linked Data describes the best practice for exposing, sharing and connecting, each piece of data (information and knowledge) that was not linked previously (or linked using other methods) in the Semantic Web. The provenance and workflow information published as Linked Data enables sharing, accessing and reuse of this information [158]. OPMW presents the provenance information of both workflow execution and abstract workflow (which, itself, contains provenance information related specific phases of workflow system lifecycle).

OPMW is designed as an OPM profile, and it extends and reuses OPM ontology and vocabulary [52]. OPMW extends OPM core concepts and relationships to be capable of capturing the execution trace of workflow templates. It also captures additional information regarding abstract templates and execution in a form of metadata which is called attribution [52]. OPMW supports the concept of roles, account and provenance graphs (which aggregate a set of OPM insertion into different sub-graphs).

In an application study using OPMW, Garijo [158] claims that current approaches to publishing workflows do not support reproducibility across different execution infrastructures. He proposed an approach to address this deficiency in the context of TB-Drugome project [160] using the OPMW model, Wings as an abstract workflow and Pegasus

(explained in section 5.2.6) as a workflow execution engine. Wings [161] is a semantic workflow system to help scientists to design computational experiments. In this approach, workflow information is published in three steps, abstract workflow, both abstract and executable workflow using OPM and OPMW, and workflow in a form of an accessible web object (Linked Data) [158].

As mentioned above, OPM is used widely among scientific workflow systems, it has a domain independent core definition, and in addition, it is an extensible model to meet such as purposes and modelling goals. The W3C provenance working group has developed the W3C Provenance Interchange Language (PROV) [51], which is a provenance standard for representing and publishing on the Web [51]. PROV [51] is ontology-based MoP similar to OPMW inspired by OPM. Provenance in PROV represents information regarding entities, activities, and people that participate in the process of producing data or thing that is used in assessing its quality, reliability or trustworthiness [51]. In the next section, the interoperability of OPM MoP is examined in the context of workflow systems.

4.4.2 Interoperability of OPM in workflow systems

In this section, we discuss some separate points regarding interoperability [53] (defined at the end of section 4.3.4) in OPM. We investigate how OPM is supported by provenance systems and discuss the difficulties of addressing interoperability.

OPM is a generic MoP, which can model the provenance of scientific workflow systems. Referring to section 3.1, every scientific workflow consists of a number of processes, (which produce or consume data), channels that connect processes together, and an interpreter (as part of workflow engine) which applies execution semantics to run the workflow, as shown in Figure 4.1. The provenance model of these scientific workflows can be modelled in OPM (as shown in Figure 4.8). There are process, artifact and agent notations in the OPM provenance model that model workflow constituent components such as process, data and a control entity (in the form of a conceptual interpreter similar to what we have explained in section 4.2.1). In addition, OPM dependencies are determined by the dependencies of processes, artifacts and agents in the workflow. OPM model can be converted into other formats such as OPMW [52,

162] and PROV [51] in order to represent provenance in different contexts like ontologies and web.

In terms of representational choice, provenance systems can represent their collected provenance information in OPM format, in a native format of workflow system or both formats [163]. In the next two paragraphs, we investigate how these formats are used in provenance systems.

Provenance mechanisms in workflow systems represent their collected provenance in OPM model (or in a format compatible with OPM) for interoperability purposes while they could represent in both native and OPM model. The native format is generally the superset of OPM model containing some information that is not presented in OPM model [163]. If provenance information is directly collected and stored just according to the format of OPM, all provenance aspects (collecting, storing, querying, presenting) use the OPM terms and semantics [163]. While, if it collects and stores according to both workflow‘s native provenance format and also OPM model at the same time, it would be possible to use terms and semantics of just the OPM format or both formats in order to satisfy different purposes and aspects of provenance information such as storing, querying or presenting.

It is possible to collect provenance in a workflow‘s native provenance format then map it into the OPM model. In this case, terms and semantics of OPM format could be used as a means to support storage and query. The mapping process could be done either during the workflow execution (during the process of provenance collection) [163], or after finishing the workflow execution (we will explore these approach in section 5.1.12). Generally, many of scientific workflow systems, including Trident, Kepler, Swift and Karma are capable of generating OPM-compliant provenance data, which is generated from their workflow’s native model [164].

The third provenance challenge [53] was organized to evaluate the interoperability of provenance systems (fourteen teams participated) in terms of importing and querying workflow provenance generated by others. The results of this challenge reveal that it is not possible for all systems to import provenance data successfully and answer all the proposed questions [154] in the third Provenance Challenges, as shown in [53, 163]. For example, the

generated OPM-compliant provenance data of Trident can successfully be imported into Kepler (COMAD) and Karma, while they can't satisfy all questions in the third Provenance Challenges, because of mapping issues (refer to end of this section) [163]. In this regard, some workflow systems, such as COMAD (in Kepler), Karma, PASS, VisTrails, enriched their OPM provenance with additional schema and data to answer the third provenance challenge's questions [154], which also causes mapping issues.

Most provenance systems supporting OPM just model retrospective provenance (regarding workflow execution), while generally, prospective provenance (which embodies the abstract workflow specification) is not modelled into OPM. This prevents OPM model from answering some of the questions that relate to workflow specifications. However, there are some approaches to address the issue of prospective provenance, such as extending OPM proposed by Lim *et al.* [165] in the context of VIEW.

The following is a list of possible reasons that prevent provenance systems from being fully interoperable in the context of workflow systems

- Provenance collection mechanisms collect provenance in various degree of details (level of abstractions), ranging from a coarse-grained provenance collected at the application level, to a fine-grained provenance collected at the platform level (explained in section 5.1.8). The diversity in the nature of collected provenance information (manifest in the form of provenance granularity) is one of the barriers for interoperability.
- The native provenance information is generally a superset of OPM model containing some information that is not presented in OPM model [163]. Mapping native provenance information to OPM may face several issues such as, losing some information that perhaps carries useful meaning; using some opaque references in native model which can't be mapped (however, some of them can be represented in a OPM profile). These issues have potential to subvert interoperability. For example, the "collection" artifact in COMAD makes use of opaque references, which cannot be interpreted and mapped into OPM graphs (however, COMAD collection can be defined in a Profile) [163].
- A lack of native support for OPM seems to be a barrier generally. Systems such as Wings [161], which have native OPM support in their design generally, can better perform interoperability (answer the questions of the third provenance challenge) than systems such as PASS which have an ad-hoc translation from their native format to OPM.
- The diversity in the tools and approaches that provenance systems use to model their provenance data and queries, cause differences in the answer to certain queries in the third

provenance challenge [154]. Workflow systems use several tools for storing and querying provenance, which can cause failure in importing provenance (or trace) files from another provenance collection mechanism using different tools. Twelve of fourteen participating teams in the third provenance challenge use different query tools and query languages. Eight of the twelve teams can import OPM data although just three of them can successfully answer some provenance questions [154].

- Workflow systems use various data and schema extensions for OPM, which violates interoperability of provenance information [154]. Six of twelve teams participating in the third provenance challenge enrich their provenance information with additional information in order to answer or improve their answer to the third provenance challenge's questions. The authors [154] do not give details of the additional information used.

In this section, the OPM and OPMW MoPs were presented in some detail. We have seen that many scientific workflow systems (some of them presented in chapter 4) derived a means of expression supporting OPM for the provenance challenge. OPM is intended to standardize and provide a widely acceptable MoP, while there are still a number of models and design issues that propose situations which may be not supported by the core OPM model as explained in this section. To address this issue the OPM Profile was introduced to make OPM extensible [50, 52, 157, 158]. We noted it is not always possible for core OPM to fully represent the provenance of systems [24], including token read and write timestamps, task parameter information, data and task annotations, support for composite tasks, and workflow specifications and modifications [24]. As a result, ontology-based MoPs (such as OPMW) are introduced to address some of the core OPM deficiencies. The extensibility of ontology-based MoP with additional terms and tags is the main advantages of these approaches.

In the next section, we explore provenance granularity and various levels of provenance granularity. As part of our discussion we link granularity to MoP. This link between granularity and MoP is the key point of the design of the MoP for our adaptive provenance collection mechanism.

4.5 A mechanism for multiple-granularity provenance

As described in section 1.3 and section 6.1, scientific workflow systems can face variety of changes during their execution. As we have seen, for many of changes, it is useful to respond

by changing the way in which provenance is collected. The response of the provenance system could be in the adjustment and adaptation of the amount of collected provenance information.

For example, as shown in Figure 1.4, workflows that have a specific budget and deadline can be run over heterogeneous computing environments. These environments contain resources whose cost of execution and cost of provenance collection vary based on demand. In workflow systems, changes in the cost of resources influence the provenance collection mechanism. For example, if the cost of provenance collection in resources is greater than the budget, the provenance collection mechanism responds to the changes in the cost of collection by collecting less provenance information. Less provenance information uses less computing and storage resources, which reduces the cost of collection.

The collection of less provenance information is enacted by reducing the level of provenance granularity (coarser grained) and collecting less detailed provenance information. As a result of this, provenance information is collected at a level of provenance granularity that is corresponds to the resources' cost and users' budgets. Thus there are various levels of provenance granularity in a provenance system facing fluctuations in its environments and requirements. The provenance system and its provenance collection mechanism should be adaptable to upcoming changes. Thus, adaptability of a provenance collection mechanism can be in the form of capturing and modelling provenance information at a suitable level of provenance granularity. The level of provenance granularity, what the provenance information is and how the provenance is represented are determined in a MoP.

As explained in section 2.1.1 fine-grained and coarse-grained provenance are two levels of provenance granularity. They have MoPs, which determine what sort of provenance is considered as fine-grained or coarse-grained provenance, and finally how they are represented. A coarse-grained MoP represents provenance information at a high level of abstraction, while a fine-grained MoP represents the detailed provenance information (in low level of abstraction). It is possible to have a variety of other MoPs for representing other levels of provenance abstraction (granularity), such as medium-grained MoP explained in 1.6.1 that represents intermediate levels of granularity.

As mentioned earlier, a provenance system benefits from having a provenance collection mechanism that is capable of adaptively collecting and representing provenance at various levels of granularity according to a MoP. This system is called an *adaptive provenance system* and its collection mechanism is called an *adaptive provenance collection mechanism*. We refer to such an *adaptive provenance collection mechanism* as a multiple-granularity provenance collection mechanism, its purpose is to collect multiple-granularity provenance information, and represent the collected provenance information based on a *multiple-granularity MoP*.

In this section, we have established the connection between the concept of MoP to the concept of provenance granularity, and investigated that a MoP represents a level of abstraction (provenance granularity) in a provenance system. Following that adaptive provenance system has introduced. In the next section, the concept of multiple-granularity provenance is explored through defining three levels of provenance granularity. We explore how these levels of provenance granularity are designed and modelled in a format compatible with OPM.

4.5.1 A design for Multiple-granularity MoP

In this section, we design three MoPs representing three levels of provenance granularity including fine-, medium- and coarse-grained. Our multiple-granularity MoP represents these levels of provenance granularity in a format compatible with OPM. This compatibility with OPM makes our MoP interoperable with other systems that use or support OPM.

As shown in Figure 4.8, OPM includes a number of elements such as Accounts, Agents, Artifacts, Processes, and Dependencies [49, 158]. An OPM description comprised of these elements can be stored in a XML file. The elements were introduced in section 4.4 and will be used in the following sections. The first element is Account [49, 158], explored in section 4.4. We use an OPM account during a collection mechanism to specify a level of granularity; each account provides a view of provenance collected at certain level of granularity. We assign an OPM account to each level of provenance granularity in a multiple-granularity MoP, as follows.

- The first account is given to coarse-grained provenance level, which provides a view toward pure OPM dependency (explained in section 4.4).
- The second account is assign to medium-grained provenance level that provides a view toward OPM dependencies and the further information about OPM relationships that are presented in a form of annotations.
- The third account is assigned to fine-grained provenance level that provides a view toward all the causal dependencies inside processes in addition to OPM dependencies and annotations on dependencies.

In the following section, we explore coarse, medium and fine-grained provenance information and their MoP. Firstly, we explore the coarse-grained provenance.

4.5.1.1 Coarse-grained MoP

The Coarse-grained MoP represents coarse-grained provenance (as explained in section 2.1.1) that is the sequence of creation of input and output data of workflow processes (workflow components). In this view, workflow processes are treated as black boxes, as shown in Figure 4.7. Coarse-grained MoP represents the history of data production that is generated in processes (such as workflow input data, workflow results, process input data, and process output) in the form of data dependencies. It also contains interactions between the workflow and external devices (such as, sensor and other data collecting equipment) and also human interaction with processes. We consider the hierarchical structure of workflow systems in the definition of coarse-grained provenance and MoP. All of OPM, Anand's Basic Model and our simple MoP can represent coarse-grained provenance information.

Figure 4.7 shows a process network including two processes (P1 and P2), process threads (PT1 and PT2) directing process executions, input ports (IP) and output ports (OP). Input tokens enter the process through input ports and output tokens are transferred to subsequent processes through output ports. In this view, there is no access to the information regarding the computational steps inside processes.

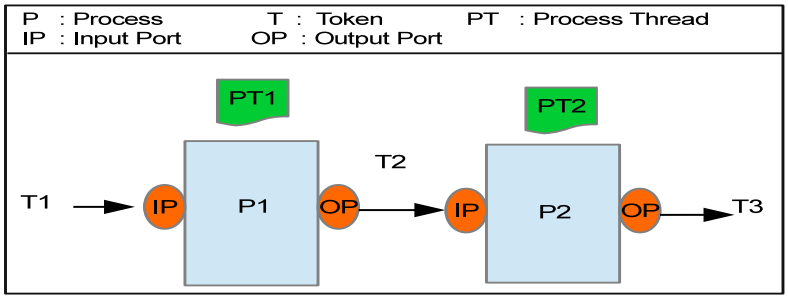


Figure 4.7.A black box view on process network graph.

Now, we model coarse-grained provenance information as an OPM model. The OPM elements (expressed in XML) of coarse-grained MoP are shown in Figure 4.8, which contains five elements including Accounts, Agents, Artifacts, Processes, and Dependencies. Each OPM element has a unique id that is used for referring to the element. Agents, artifacts and processes have a label in addition to the id and account. The label indicates another reference to OPM elements. For example, we use the names of processes as their label. We assign an Id to the account of coarse-grained MoP (account id =1 is set for coarse-grained MoP in our implementation). This account is repeated in all the elements in the structure of OPM and coarse-grained MoP in order to provide a view for coarse-grained provenance. Therefore, it is possible to refer to a view of coarse-grained provenance using its account's identifier.

As shown in Figure 4.8, the dependency section in OPM structure contains five possible dependencies (described in section 4.4) including WasTriggeredBy (WTB), Used, WasGeneratedBy (WGB), WasDerivedFrom (WDF) and WasControlledBy (WCB). These dependencies have id, effect, cause and account as shown in Figure 4.9. Logically, a cause results in an effect. Therefore, in the dependency, an effect has dependency to a cause. For example, in a WGB dependency shown in Figure 4.9 in a process (P2) that generated a token (T3), the token is an effect of a process. Thus, process (P2) is a cause of existence of that token (T3) in a WGB dependency. The OPM dependencies of Figure 4.9 are shown in Table 4.1, including effect, OPM dependency and cause. For example, P1 cause triggering P2, which is shown in a form of WasTriggeredBy dependency. Therefore, P2 was triggered by P1 as shown in the Table 4.1.

Coarse-grained OPM structure:

- Accounts: Id
- Agents: Id, Account, Label
- Artifacts: Id, Account, Label
- Processes: Id, Account, Label
- Dependencies:
  - WTB
    - Id, Effect, Cause, Account
  - WGB
    - Id, Effect, Cause, Account
  - USED
    - Id, Effect, Cause, Account
  - WDF
    - Id, Effect, Cause, Account
  - WCB
    - Id, Effect, Cause, Account

Figure 4.8.OPM structure for coarse-grained MoP.

```
<wasGeneratedBy id="WGB_2">
    <effect ref="T_3"/>
    <cause ref="P_2"/>
    <account ref="account_1"/>
</wasGeneratedBy>
```

Figure 4.9. WasGeneratedBy dependency in coarse-grained provenance.

Table 4.1. OPM dependencies for coarse-grained MoP.

Effect	OPM Dependency	Cause
P2	WTB	P1
P1	Used	T1
P2	Used	2
T2	WGB	P1
T3	WGB	P2
T2	WDF	T1
T3	WDF	T2
P1	WCB	PT1
P2	WCB	PT2

The coarse-grained MoP just establishes the basic OPM elements and models basic OPM dependencies. In the next section, we introduce medium-grained provenance and its MoP.

4.5.1.2 Medium-grained MoP

Medium-grained provenance information is a refinement (defined in the section 4.4) of coarse-grained provenance because the medium-grained provenance information has coarse-grained provenance information in addition to extra information shown in the form of annotation elements of OPM [49]. These annotations are the main difference between our coarse- and medium-grained MoP. The OPM structure for medium-grained provenance is shown in Figure 4.10 that some annotations includes

- Data type and data values to artifacts
- Role assigned to WGB, Used and WCB dependencies.
- A reference to the token that causes WTB dependency
- A reference to the process that drives WDF dependency.

We assign an unique account Id (account id =2) to medium-grained provenance in our OPM model. The view of provenance information presented by the medium-grained account is a more complete description of the view presented by a coarse-grained account. The dependencies in medium-grained provenance for the process network shown in Figure 4.10 are presented in Table 4.2.

The medium-grained MoP adds some extra information in the form of annotations (as listed in Table 4.2) to basic OPM element (artifact) and models basic OPM dependencies. The extra information makes the MoP more informative. For example, it provides information about a process that is involved in derivation of tokens in a WDF dependency. In the next section, we introduce fine-grained provenance and its MoP.

Medium-grained OPM structure:

- Account: id
- Agent: Id, Account, Label
- Artifacts: Id Account, Label+ **Token Value, Token Type**
- Process: Id, Account, Label
- Dependency:
    - WTB
        - Id, Effect, Cause, Account, **Token Id**
    - WGB
        - Id, Effect, Cause, Account, ***Role***
    - USED
        - Id, Effect, Cause, Account, ***Role***
    - WDF
        - Id, Effect, Cause, Account, **Process Id**
    - WCB
        - Id, Effect, Cause, Account, ***Role***

Figure 4.10.OPM structure for medium-grained MoP.

Table 4.2. OPM dependencies for medium-grained MoP.

Effect	OPM Dependency	Cause	Annotation
P2	WTB	P1	Token id : T2
P1	Used	T1	Role
P2	Used	T2	Role
T2	WGB	P1	Role
T3	WGB	P2	Role
T2	WDF	T1	Process id: P1
T3	WDF	T2	Process id: P2
P1	WCB	PT1	Role
P2	WCB	PT2	Role

4.5.1.3 Fine-grained MoP

As mentioned in section 4.1, we refer to the provenance conventionally collected by workflow systems as course-grained provenance. However, there are some advantages in having fine-grained provenance. In the literature there are diverse meanings attached to the term "fine-grained provenance". Therefore, we will clarify an interpretation of the term, in the next section. We consider the hierarchical structure of scientific workflow system in order to

define fine-grained provenance, as a main significant difference between our definition and others. Fine-grained provenance (known as data provenance in database community [76, 166]) is represented the provenance of data captured at the system level (at a system or database level) regardless of the structure of system, as we explore in section 4.1. The fine-grained provenance in system level and database level represents the finer grained data and derivation of data in comparison with our definition of fine-grained provenance. In this section, we will make explicit the concept of fine-grained provenance in scientific workflow systems. We will explore the advantages of having fine-grained provenance and also we will provide a MoP for fine-grained provenance.

4.5.1.3.1 Fine-grained provenance information

As explored in section 2.1.1 and in previous section, fine-grained provenance in the literature [20, 24, 75, 76] is referred as the sequence of creation of data in system's components at the database level. While fine-grained provenance, in our thesis, represents provenance information based on the hierarchical structure of workflow system. Fine-grained provenance represents provenance information about the interaction both inside and between workflow components. Fine-grained provenance information is a refinement of coarse-grained and medium-grained provenance information. It contains all information and dependencies defined in coarse-grained provenance; in addition to some additional (fine-grained) provenance information presented in the form of data-, process- and data-process dependency (explained in section 4.2.1). The additional provenance information in fine-grained provenance includes intermediate data produced and consumed inside processes; and computational steps applied to intermediate data (intermediate dependency) resulting in the process outputs and workflow results [167]. Fine-grained MoP enables workflows to disclose (data-, process-, and data-process) causal dependencies inside processes, method calls, and intermediate task (method) dependencies inside workflow processes.

Figure 4.11 presents the process network shown in Figure 4.7 in a way that intra-process activities are accessible and observable, similar to white-box approach in software engineering. In this Figure, intermediate data and methods are shown inside processes unlike the Figure 4.7 to show intermediate entity and interactions that are important for fine-grained provenance.

The methods in processes appear in a certain ordering (linearized in time ordering) which is a linearization of a partial ordering defined by data dependencies between P1 and P2; or between methods in either processes. The data dependencies between P1 and P2 are expressed through the mechanism of sending and receiving data (such as ports and channels). These send and receive mechanisms determine ordering. For instance in Figure 4.11, the order of method execution (shown as a linearization of the partial ordering of method calls) shows P1 appears before P2. In the partial ordering the invocation of methods in P1 precedes the invocation of the corresponding method in P2. For example, as shown in Figure 4.11 the MA() method of P1 appears before MD() method of P2 in trace of execution (If MA() in P1 is the corresponding method of MD() in P2). The dependency of P2 on P1 implies that there is an ordering MA() before MD(). This dependency is characterized in terms of OPM.

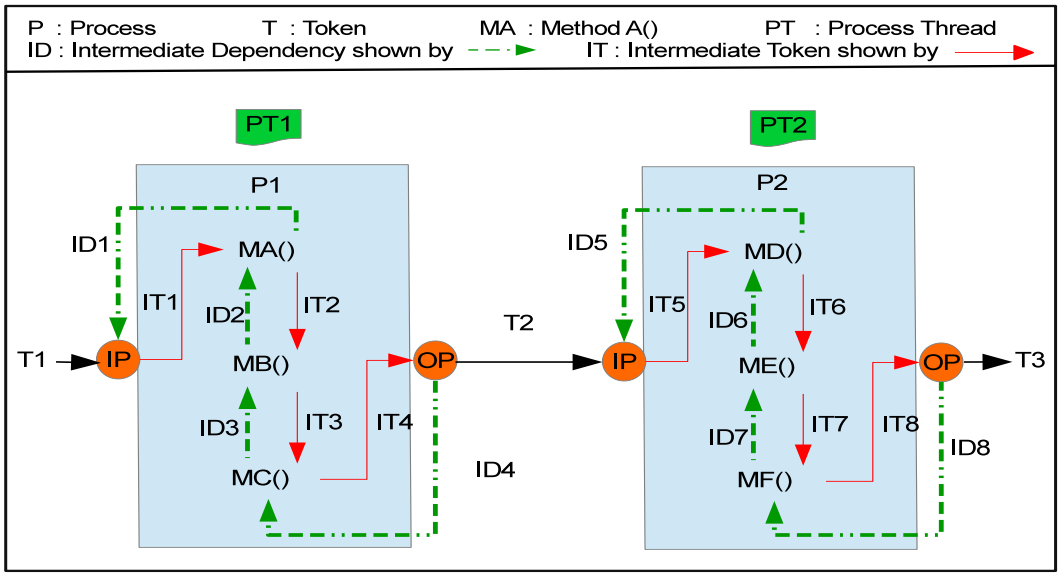


Figure 4.11.A white box view on process network graph with intra-process activities.

Fine-grained provenance are also concerned with the dependencies between methods inside processes: we explore these dependencies now. In a classic execution-time dependency that a method (such as MA() in P1) executed before the execution of another method (such as MB() in P1). The latter method MB() should not happen before the execution of MA(). Therefore, the first method (MA()) causes the execution of latter method (MB()). This is dynamic execution chain (relationship) that is considered in the collection of provenance dependency. The dynamic execution chain (inside any processes) would be sequential time-ordered list of methods. In this dynamic execution chain, the execution of each method was caused by execution of another method. For example, the execution MB() in P1 was caused by

execution of MA() in P1 (causality relationship); however, MB() does not necessarily invoked by MA(). In Figure 4.11, MB() Was indirectly invoked by MA() or MA() causes the execution of MB(). Moreover, in the dynamic execution chain the existence of a data (token or artifact) is dependent on predecessor data.

Generally, software architectures are concerned with coarse-grained MoP and consider processes and components as black-boxes [47, 77, 166, 167]. The capturing provenance information is manageable for provenance systems (in black-boxes systems) during the execution phases. However, recording fine-grained provenance is not a trivial issue. It requires understanding regarding design and implementation details of underlying Model of Computation (MoC) and the structure of workflow entities. In order to have this understanding of design and implementation the computational steps such as methods, and data products inside processes should be disclosed (known as "white-box" approach) [167] as shown in Figure 4.11. A provenance collection mechanism, aiming to capture fine-grained provenance, should have access to design and implementation details of the underlying MoC and the structure of workflow components, in order to construct fine-grained provenance including intermediate tokens and dependencies, which increase the size of collected provenance information.

As shown in [168], in some cases, the size of collected fine-grained provenance model is more than the size of the input or output dataset. In the case study presented in Figure 4.11, an input token has an output token, while the amount of provenance information in Figure 4.12 and particularly in Table 4.3 is far more than single input and output token. Malik *et al.* [168] also refered to this point that fine-grained provenance can have high storage overheads, with consequences on performance if the data needs to be sent over a network.

4.5.1.3.2 Why Fine-grained provenance

Glavic *et al.* [169, 170] proposed a number of case studies that require fine-grained provenance. Although the case studies are in the area of stream processing, it is quite similar to the concept that is used in scientific workflow systems, in terms of processing a sequence of input data and generating a sequence of output data.

- Fine-grained provenance information can assist in exploring the steps in stream query debugging and diagnosis in a way that tracing the output result back to the input data to realize where an error happens and how it has propagated.
- Fine-grained provenance information can assist in revealing how events become part of a data warehouse for mining and analysis with the help of fully collected provenance while event warehousing was used to collect raw and derived event stream for this purpose.

It is productive to capture intra-process provenance (similar to fine-grained provenance information shown in Table 4.3) to understand what computational steps occurred inside processes and how data are produced. Fine-grained provenance information can enrich scientific workflow systems including enhancing the workflow reproducibility. Scientific workflow systems thus benefit from fine-grained provenance information in order to reproduce precisely and also simulate the scientist's experiments. Reproducibility is one of the main requirements of SWfMSs as explained in section 5.2.1. It enables scientists to retrace workflow experiments, which they were not involved in, and enables them to repeat experiments and improve the accuracy of their work. SWfMSs could have a component to rerun the experiment or simulate the execution of the experiment with the help of collected provenance information from the experiment. Reproducibility with the coarse-grained provenance information could show what the outputs of processes are for the given inputs. But it does not reveal how activities generate these outputs based on the inputs (what happen inside processes).

As explored earlier, some workflow systems wrap components and consider them as black boxes [77] therefore, a provenance system mostly inspects the input and output of these wrapped components instead of looking inside to realize how data is created. Fine-grained provenance such as intermediate dependencies (explored in section 4.5.1.3.1) provides more precise representation of scientific workflows. Tracking intermediate data allows recording of finer-grained behaviour [171]. Fine-grained provenance can enhance the process of determining the quality and validity of scientific data, which are very important in standardizing provenance and developing provenance query frameworks [167]. Fine-grained provenance with more detailed information about creation of data helps the growth of fidelity and trustworthiness [77] of provenance systems in scientific workflow system.

So far, we have explored the concept, elements and importance of fine-grained provenance information. In the next section, a MoP is explained for the fine-grained provenance information.

4.5.1.3.3 A Fine-grained MoP

In this section, we describe a fine-grained MoP compatible with structure of OPM, and we show an example of (the dependencies of) fine-grained provenance in Table 4.3 for the process network presented in Figure 4.11. Our example of fine-grained provenance is based on the fine-grained MoP shown in Figure 4.12. The example includes a mapping mechanism between the workflow and OPM that is presented as the concept of an OPM Profile [49] defined in section 4.4.

Figure 4.12 shows an OPM file structure for a fine-grained MoP. In this OPM structure, intermediate tokens (data) (such as IT1 and IT2 in P1) are added to artifact's elements; and intermediate methods (such as MA() and MB() in P1) are added to process elements in the form of annotations that contain id, account and label. The input and output ports are seen as methods (named IP and OP method respectively) from the interactions point of view inside processes. We assign a unique account Id (account id =3) to fine-grained provenance in our OPM model shown in Figure 4.12. The view of provenance information presented by the fine-grained account is a more complete description of the view presented by medium-grained and coarse-grained accounts.

Intermediate-tokens and intermediate-methods causing intra-process dependencies are shown under the dependency element in Figure 4.12. These dependencies contain id, effect, cause, account and process id. The process id refers to the process whose dependency happens inside it. As shown in Figure 4.12 intra-process dependencies include

- *Intra-process WasIndirectlyInvokedBy (I-WIIB)* (or Intra-process WasTriggeredBy (I-WTB)) reveals the causal dependency of a method (MB() in P1) to another method (MA() in P1), indicating a method (MB() in P1) was indirectly invoked by another method (MA() in P1) or a method (MA()) causes the execution of another method (MB()).

- *Intra-process WasGeneratedBy (I-WGB)* reveals the causal dependency of one or a number of artifacts (IT3 in P1) to a method (MB() in P1), indicating a method (MB() in P1) execution was required to be initiated for the artifact(s) (IT3 in P1) to have been generated.
- *Intra-process Used (I-Used)* reveals the causal dependency of a method (MB() in P1) to one or a number of artifacts ((IT2 in P1)), indicating completion of method execution (MB() in P1) were required to the availability of artifact(s) (IT2 in P1).
- *Intra-process WasDerivedFrom (I-WDF)* reveals the causal dependency of an artifact (IT3 in P1) to another artifact (IT2 in P1), indicating an artifact (IT2 in P1) requires to have been generated for artifact (IT3 in P1) to be generated. Artifact (IT3 in P1) generating method was derived from presence of (predecessor) artifact (IT2 in P1).

Fine-grained OPM structure:

- Agent: Id, Account, Label
- Artifacts: Id Account, Label, **Token Value, Token Type**
    - **Intermediate Artifacts (Token): Id, Account, Label, Token Value, Token Type**
- Process: Id, Account, Label
    - **Intermediate Method: Id, Account, Label**
- Dependency:
    - WTB
        - Id, Effect, Cause, Account, **Token Id**
    - WGB
        - Id, Effect, Cause, Account, ***Role***
    - USED
        - Id, Effect, Cause, Account, ***Role***
    - WDF
        - Id, Effect, Cause, Account, **Process Id**
    - WCB
        - Id, Effect, Cause, Account, ***Role***
    - I-WIF
        - Id, Effect, Cause, Account, ***Process Id***
    - I-WGB
        - Id, Effect, Cause, Account, ***Process Id***
    - I-USED
        - Id, Effect, Cause, Account, ***Process Id***
    - I-WDF
        - Id, Effect, Cause, Account, ***Process Id***

Figure 4.12. OPM structure for fine-grained MoP.

Fine-grained MoP is superset of medium and also coarse-grained MoP. Table 4.3 presents dependencies of fine-grained MoP for the process network shown in Figure 4.11. The fine-

grained MoP includes medium-grained (and consequently coarse-grained) dependencies presented in a form of basic OPM dependencies (WTB, WGB, Used, WDF and WDF) and intra-process dependencies in processes (both P1 and P2) presented in a form of I-WIIB, I-WGB, I-Used and I-WDF. The first section of Table 4.3 is similar to Table 4.1 and 4.2, which are not revealing the creation and existence of data inside a process. As shown in Figure 4.12 and Table 4.3, fine-grained MoP has intermediate tokens and methods inside processes and some intra-process dependencies in addition to medium-grained provenance information.

The fine-grained information presented in fine-grained MoP contains detailed information explaining how data comes into existence inside processes (as shown in Table 4.3), which makes the provenance more complete in comparison with the coarse-grained provenance. The coarse-grained provenance is not able to explain how data come into existence inside processes because it does not have access to methods inside processes that produce and consume data.

Table 4.3 aims to show how dependencies involving fine-grained provenance could be presented, particularly the dependencies inside processes. This Table shows intra-process dependencies including I-WGB, I-WIIB, I-Used and I-WDF, in addition to standard OPM dependencies including WTB, USED, WGB, WDF and WCB. The intra-process dependencies do not exist in coarse-grained provenance because of restricted access to data and methods inside processes. For example, a I-WGB dependency shown in eleventh row (the second row of second section) of Table 4.3, represents a dependency between an intermediate token (IT2) and a method (MA()). The method appears inside a process (P1) as shown in annotation column. This I-WGB dependency illustrates that IT2 is created by the MA() method inside P1 process.

In the rest of this section, we explore how we make the format of our MoPs compatible with OPM. We have expressed how a compatible MoP with OPM can model coarse, medium and fine-grained provenance information. The core OPM notations can express coarse-grained provenance information (as we presented in Figure 4.8 and Table 4.1) and make a MoP compatible with OPM. OPM defines extra information in elements of OPM structure in the form of annotations [49] as shown in Figure 4.10 and 4.12. These annotations are the main

difference between coarse- and medium-grained MoP. The annotations on OPM dependencies in medium- and fine-grained MoP are shown in Table 4.2 and 4.3.

Table 4.3. OPM dependencies for fine-grained MoP.

Effect	OPM Dependency	Cause	Annotation
P2	WTB	P1	Token id : T2
P1	USED	T1	Role
P2	USED	T2	Role
T2	WGB	P1	Role
T3	WGB	P2	Role
T2	WDF	T1	Process id: P1
T3	WDF	T2	Process id: P2
P1	WCB	PT1	Role
P2	WCB	PT2	Role
IT1	I-WGB	IP method	Process id: P1
IT2	I-WGB	MA()	Process id: P1
IT3	I-WGB	MB()	Process id: P1
IT4	I-WGB	MC()	Process id: P1
MA()	I-USED	IT1	Process id: P1
MB()	I-USED	IT2	Process id: P1
MC()	I-USED	IT3	Process id: P1
OP method	I-USED	IT4	Process id: P1
MA()	I-WIIB	IP method	Process id: P1
MB()	I-WIIB	MA()	Process id: P1
MC()	I-WIIB	MB()	Process id: P1
OP method	I-WIIB	MC()	Process id: P1
IT1	I-WDF	T1	Process id: P1
IT2	I-WDF	IT1	Process id: P1
IT3	I-WDF	IT2	Process id: P1
IT4	I-WDF	IT3	Process id: P1
T2	I-WDF	IT4	Process id: P1
IT5	I-WGF	IP method	Process id: P2
IT6	I-WGF	MD()	Process id: P2
IT7	I-WGF	ME()	Process id: P2
IT8	I-WGF	MF()	Process id: P2
MD()	I-USED	IT5	Process id: P2
ME()	I-USED	IT6	Process id: P2
MF()	I-USED	IT7	Process id: P2
OP method	I-USED	IT8	Process id: P2
MD()	I-WIIB	IP method	Process id: P2
ME()	I-WIIB	MD()	Process id: P2
MCF()	I-WIIB	ME()	Process id: P2
OP method	I-WIIB	MF()	Process id: P2
IT5	I-WDF	T2	Process id: P2
IT6	I-WDF	IT5	Process id: P2
IT7	I-WDF	IT6	Process id: P2
IT8	I-WDF	IT7	Process id: P2
T3	I-WDF	IT8	Process id: P2

There are a number of elements including intermediate methods, data and their dependencies in the fine-grained MoP (shown in Figure 4.12 and Table 4.3) that make the MoP inconsistent with core OPM. The new elements and concepts need to be formalized in an OPM Profile [49], as explained in section 4.4. We present here a Profile as a mapping mechanism between

workflow and OPM to provide an explicit mapping for intermediate methods, data and dependencies, as shown in Table 4.4.

The profile presented in Table 4.4 shows the mapping of workflow events to OPM entities. Process Network Application (PNA) as explained in section in chapter 3 gives us an implementation of a workflow. Therefore, we show PNA events as modelling workflow events. The Table shows the way events generated in PNA cause the creation of an OPM graph, processes, agents, artifacts and dependencies. The profile makes the mapping explicit between PNA entities and OPM entities. In our design, we can characterize the mapping mechanism of the Profile as either to OPM or the MoP (presented in this section) because the MoP is defined in an OPM consistent manner.

Table 4.4. OPM-Profile-Mapping OPM entities during workflow (PNA).

Workflow Event	**Actions**
Workflow started	- Instantiate OPM graph - Create Account Ac_i and initialize based on requested granularity- Ac3 for fine-grained-Ac2 for medium and Ac1 for coarse-grained
Process P as a part of workflow system started	- Create process P
Trigger method in Process Thread PT started	- Create Agent Ag - Create WasControlledBy (R, Ag, P) - Fetch Account Ac_i and add to all P, Ag and WasControlledBy
Data entered in Input Port IP in Process P	- Create Artifact A_i, If it is new Artifact - Create USED (IP_i, P, A_i) for each Input Port IP_i - Create WasTriggeredBy (P_i, P_j)for each P_i to each P_j as processor Process - Fetch Account Ac_i and add to all A_i and USED and WasTriggeredBy
Data get out of Output Port OP in Process P	- Create Artifact A_i , If it is new Artifact - Create WasGeneratedBy (OP_i, P, A_i) for each Output Port OP_i - Create WasDerivedFrom (A_i, A_j) for each Artifacts A_i in each Output Port OP_i to each A_j in each Input Port IP_i - Fetch Account Ac_i and add to all A_i, WasDerivedFrom and WasGeneratedBy
Intermediate method (only fire method) created data	- Create Artifact A_i (intermediate-artifact annotation, Process p) - Create WasDerivedFrom (A_i, A_j,I-WDF annotation, Process p) for each produced Artifacts A_i from each consumed Artifacts A_j in Process P - Create WasGeneratedBy (OP_i, P, A_i, I-WTB annotation, Process P) for each produced Artifact A_i as a return value of a method P. - Fetch Account Aci for fine-grained and add to all A_i, WasDerivedFrom and WasGeneratedBy
Intermediate method invocation	- Create process P (intermediate-method annotation, Process p) - Create USED (IP_i, P, A_i, , USED annotation, Process P) for each used Artifact A_i as method parameter in each method P, In case there is a parameter. - Create WasTriggeredBy (P_i, P_j, I-WTB annotation, Process P) for each(called) method P_i to each (caller) method P_j in Process P. {It is defined as I-WIIB dependency) - Fetch Account Aci for fine-grained and add to all A_i, USED and WasTriggeredBy

In chapter 7, we follow the profile's events and actions in order to generate a provenance file; however, we do not implement the profile literally. In our implementation, the intermediate data, methods and dependencies are annotated with "intermediate" under corresponding OPM elements (see appendix A), instead of being implemented with the profile concept described here. For example, intermediate artifact is constructed under the artifact element in OPM

graph with an “intermediate” annotation; and intermediate WasDerivedFrom dependencies (I-WDF) as an example are constructed under the dependencies element in OPM graph and in a form of OPM WasDerivedFrom dependency with an “intermediate” annotation.
This type of implementation also makes our implementation consistent with the conventional view of fine-grained provenance in other communities such database; however, our definition of fine-grained provenance, explored in section 4.5.1.3, is different for other definitions by considering where the data and methods (tasks) are happen in workflow structure. As an example, we treat data and methods (tasks) inside processes differently to data and methods (tasks) outside processes. However, all the data are represented as OPM artifacts; and all the methods and tasks (every computational step) are represented as OPM processes. Our implementation makes generated file compatible with OPM notation and general perception of it, against the differences that exist in our definition of fine-grained provenance.

4.5.2 Discussion of multiple-granularity provenance

So far, we have explored provenance information and MoP in terms of dependencies. However, there are redundant dependencies (such as WasTriggeredBy) that we will formulate below. We will identify what are redundant dependencies in the context of data-oriented and process-oriented provenance, in order to take them out from our MoP. We will also customize the presented multiple-granularity provenance in previous section to make it concise and to conform to what users of scientific workflows want.

Workflow systems are categorized in terms of scientific and business workflow systems as explored in section 2.3. The scientific (or dataflow) workflow is data-driven because processes in scientific workflow are activated by data, which is called data-oriented [44]. The flow of data within or between scientific workflow’s processes is expressed by dependencies. The data-oriented provenance is characterised by WasDerivedFrom (WDF) dependency of OPM concerning the causal dependency between artifacts. Therefore, provenance in scientific workflow system is represented by WasDerivedFrom (WDF) dependency, and it is not necessary to collect some redundant dependencies such as WasTriggeredBy (WTB), Used and WasGeneratedBy (WGB).

A business workflow is control-driven because processes in business workflow are activated by control dependencies and the defined transition condition over them. It represents the control dependencies and partial ordering relation of (business) workflow activities [172-174], which is called process-oriented [44] provenance. Process-oriented provenance reveals the sequence of process execution, which is represented by WasTriggeredBy (WTB) dependency concerning the causal dependency between processes. Other dependencies are the redundant dependencies, which are not necessary to be collected.

Therefore, a MoP can be customized to collect and represent the only provenance information that is required (not the redundant dependencies). The customization and reductions of provenance dependencies make provenance simpler with a smaller amount of provenance information but still meaningful.

There is another mechanism, defined in OPM that can help in reducing the number of extra dependencies. This mechanism is based on inference mechanisms on OPM dependencies as mentioned in [49, 75, 159, 175] (referred as completion rules in [49]). The inference model shown in Figure 4.13, either of WDF and WTF dependencies can be deduced from Used and WasGeneratedBy (WGB) dependencies. Therefore, a provenance collection mechanism can collect Used and WGB dependencies and satisfy both data-oriented and process-oriented provenance using the inference mechanism. In this mechanism instead of collecting all the OPM dependencies, the Used and WGB are constructed and, from these, other dependencies can be constructed as required.

In Figure 4.13, there is a process (P1) that used an artifact (A1) (Used dependency) and generates another artifact (A2) (WGB dependency). In this Figure, the used and generated artifacts (in Used and WGB dependencies) establish a WDF dependency between these two artifacts. This WDF can be inferred from collected Used and WGB dependencies.

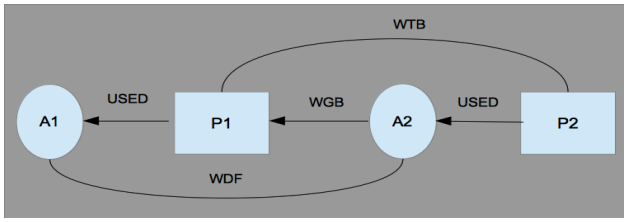


Figure 4.13. One Step Inference in the Provenance Model [93, 175].

For example, the collected provenance dependencies in coarse-grained MoP can be reduced to Used and WGB presented in as shown in Figure 4.14 and Table 4.5. This form of coarse-grained MoP is used when users want to have data, process and data-process causal dependencies.

OPM structure:

- Accounts: Id
- Agents: Id, Account, Label
- Artifacts: Id, Account, Label
- Processes: Id, Account, Label
- Dependencies:
  - WGB
    - Id, Effect, Cause, Account
  - USED
    - Id, Effect, Cause, Account
  - WCB
    - Id, Effect, Cause, Account

Figure 4.14. OPM structure for coarse-grained MoP.

Table 4.5. OPM dependencies for coarse-grained MoP.

Effect	OPM Dependency	Cause
P2	Used	T2
P1	Used	T1
T2	WGB	P1
T3	WGB	P2
P1	WCB	PT1
P2	WCB	PT2

We are not interested in the first form of reduction in coarse-grained MoP in this thesis, that is, collecting WGB and Used dependency, then inferring the WasTriggeredBy (WTB) and WasDerivedFrom (WDF) dependencies from them. It seems more efficient to capture just the WasTriggeredBy (WTB) dependency, if a provenance system is concerned with causal relationship between processes. Similarly if a provenance system is concerned with creation of data, then it captures just the WDF dependency, as shown in Figure 4.15 and Table 4.6. In the rest of this section, we will explore data-oriented coarse, medium and fine-grained MoP.

Figure 4.15 presents a data-oriented coarse-grained MoP containing all the required OPM elements. It is similar to coarse-grained MoP presented in Figure 4.8 but it has just WasDerivedFrom (WDF) and WasControlledBy (WCB) under the dependency element. The dependencies of data-oriented coarse-grained provenance according to Figure 4.7 are presented in Table 4.6. The MoP shown in Figure 4.15 presents a minimal but meaningful and useful form of coarse-grained MoP called data-oriented coarse-grained MoP. The WCB dependency is presented in both situations mentioned (either data or process-oriented MoP). It is also possible to present both of WDF and WTB dependency together, if users are looking forward to record dependency of both data (artifacts) and processes.

Data-oriented coarse-grained OPM structure:

- Accounts: Id
- Agents: Id, Account, Label
- Artifacts: Id, Account, Label
- Processes: Id, Account, Label
- Dependencies:
  - WDF
    - Id, Effect, Cause, Account
  - WCB
    - Id, Effect, Cause, Account

Figure 4.15. OPM structure for Data-oriented coarse-grained MoP.

Table 4.6. OPM dependencies for Data-oriented coarse-grained MoP.

Effect	OPM Dependency	Cause
T2	WDF	T1
T3	WDF	T2
P1	WCB	PT1
P2	WCB	PT2

These two forms of reduction in the number of provenance dependencies make coarse-grained provenance simpler with a smaller amount of provenance information whilst still retaining meaning. In this thesis, we are interested in having data-oriented coarse-grained provenance shown in Figure 4.15 and Table 4.6. The medium-grained MoP in our implementation would be the same as designed in section 4.5.1.2; however, it is possible to have some reduction in

the number of collected dependencies similar manner to the reduction for coarse-grained MoP.

A similar reduction also applies to fine-grained provenance. The fine-grained MoP should be designed such as a way to address the requested provenance information (either data-oriented or process-oriented) and ignore extra dependencies. Moreover, it will still be a refinement of coarse and fine-grained MoP. For example, if users were interested in data-oriented fine-grained provenance, the provenance collection mechanism would collect five basic OPM dependencies and also intermediate data and I-WDF intra-process dependency. Collecting the basic OPM dependencies makes it a refinement of coarse and fine-grained MoP, and the WDF and I-WDF makes it data-oriented.

Fine-grained OPM structure:

- Agent: Id, Account, Label
- Artifacts: Id, Account, Label, Token Value, Token Type
  - Intermediate Artifacts (Token): Id, Account, Label, Token Value, Token Type
- Process: Id, Account, Label
  - Intermediate Token: Id, Account, Label
- Dependency:
  - WTB
    - Id, Effect, Cause, Account, Token Id
  - WGB
    - Id, Effect, Cause, Account, *Role*
  - USED
    - Id, Effect, Cause, Account, *Role*
  - WDF
    - Id, Effect, Cause, Account, Process Id
  - WCB
    - Id, Effect, Cause, Account, *Role*
  - I-WDF
    - Id, Effect, Cause, Account*, Process Id*

Figure 4.16. OPM dependencies for Data-oriented fine-grained MoP.

Figure 4.16 shows the OPM structure for data-oriented fine-grained MoP. In this OPM structure for data-oriented fine-grained, intermediate tokens (such as IT1 and IT2 in P1) are added to artifact's element, in addition to all the required OPM elements (including Accounts, Agents, Artifacts, Processes, and Dependencies). The dependency element of this data-

oriented fine-grained MoP (shown in Figure 4.16) includes an intra-process dependency between intermediate tokens (artifacts) inside processes (I-WDF), in addition to other OPM dependencies. Table 4.7 shows the dependencies of data-oriented fine-grained provenance according to Figure 4.16 that includes the I-WDF dependencies inside P1 and P2.

Table 4.7. OPM dependencies for Data-oriented fine-grained MoP.

Effect	OPM Dependency	Cause	Annotation
P2	WTB	P1	Token id: T2
P2	Used	T2	Role
P1	Used	T1	Role
T2	WGB	P1	Role
T3	WGB	P2	Role
T2	WDF	T1	Process id: P1
T3	WDF	T2	Process id: P2
P1	WCB	PT1	Role
P2	WCB	PT2	Role
IT1	I-WDF	T1	Process id: P1
IT2	I-WDF	IT1	Process id: P1
IT3	I-WDF	IT2	Process id: P1
IT4	I-WDF	IT3	Process id: P1
T2	I-WDF	IT4	Process id: P1
IT5	I-WDF	T2	Process id: P2
IT6	I-WDF	IT5	Process id: P2
IT7	I-WDF	IT6	Process id: P2
IT8	I-WDF	IT7	Process id: P2
T3	I-WDF	IT8	Process id: P2

4.6 Summary

In this chapter, we have elaborated the concept of MoP as a way of determining what provenance information should be collected and how they are represented. Many provenance systems are modelled and designed assuming a simple construction for workflow constituents involved in provenance (such as process and data) [24, 142] . In this manner such systems are similar to our simple MoP. However, the workflow systems of these provenance systems are often augmented by advanced features and structures to deal with non-atomic data structure and processes with multiple task invocations (as explored in section 4.2.1). The MoP needs to

be compatible with the features and structures presented in workflow systems. Consequently, workflow systems with more complex MoC (such as involving distributed scheduling, streaming and concurrency) [24] and also more complex architecture and structure (such as involving running on remote and distributed hosts, different transformation approaches in workflow channel possible in distributed environments and nested data structure) require more complex MoP with additional provenance modelling construction to collect and present provenance information.

We suggest that a MoP would be benefit from being defined and designed based on requirements and applications of provenance systems, in addition based on the structure of workflow system that its provenance information is presented in the MoP. We have reviewed a number of MoPs, namely Anand [142], COMAD [142, 152], Muniswamy-Reddy [72], OPM [49, 153], and OPMW [52]. A number of MoPs includes Anand, COMAD and Muniswamy-Reddy are explored to demonstrate the structure of provenance information and various dependencies patterns among them, while, OPM is explored to demonstrate the representation model of a MoP. OPM is proposed to tackle the interoperability issue in provenance systems. Therefore, various MoPs could be designed, ranging from a simple MoP to a sophisticated one similar to COMAD. The Anand [142] and Muniswamy-Reddy [72] systems are two examples that have several MoPs that are representative of provenance information captured form workflow systems with different features and structures.

We have explored the principal concepts and interoperability of the OPM MoP. OPM and its key concepts such as dependencies, Profile, Role, Account and Annotation are defined and also used in our MoP design in this chapter. OPM is supported in many scientific workflow systems, which we will discuss and present a number of them in chapter 4.

We have proposed our definition of coarse-, medium and fine-grained level of provenance granularity for scientific workflow systems. We have made a connection between a MoP and level of provenance granularity and have proposed a MoP for provenance collection mechanisms capable of capturing various levels of provenance granularity as a result of fluctuation in environments and requirements. We propose the MoP aims to represent multiple-granularity provenance information used in our design and implementation in later

chapters. Finally, we have investigated some customization the multiple-granularity MoP to makes it concise and suitable for scientific workflow systems.

We are going to use the concept of MoP in subsequent chapters in this thesis, importantly, using this concept in chapter 5 as one of the classification specifications for workflow systems and we will also implement the OPM MoP in our implementation in chapter 7.

5 MECHANISMS FOR PROVENANCE COLLECTION

Scientific workflow management systems run scientific experiments. They manage sequences of complex process transformations and collect provenance information at various levels of abstraction. Collected provenance information from scientific experiments declares how experimental results are derived from input values along with experimental parameters and workflow configurations. Provenance helps turn workflow systems into widely acceptable systems among scientists, because provenance allows workflow systems to capture process configuration and behaviour at different levels of detail [26]. On this basis, a sufficient level of collected provenance information enables scientists to validate their hypotheses and make the workflow reproducible. Currently SWfMS's do not use a standard or portable provenance model for either capturing, storing, querying or representing model [7]. There are a variety of design issues in provenance models and mechanisms in workflow system, owing to the variation of design dimensions in workflow architectures. Given this variety, it seems necessary to classify the provenance mechanisms in workflow systems.

In this section, different design dimensions and conventions for provenance collection mechanisms are surveyed in the context of scientific workflow systems. Then, those conventions are used in order to evaluate provenance collection mechanisms, presented at the end of this chapter

5.1 Introduction

Recall from section 2.1.1, the concept of provenance is associated with the sequence of creation of data [45]. Provenance captures information about dependencies between data and processes [21]. Our focus here is on provenance as used to capture information about scientific workflow systems. In scientific workflows, provenance is a (dynamic) mechanism that collects and manages sufficient levels of static and dynamic workflow information in order to enable reproduction, inspection and validation of experimental results

As mentioned in the introduction, provenance is a key asset in workflow systems and SWfMS. Many novel contributions have been recently added to the concept of provenance (including major developments in representation models of provenance [49-53] and in

provenance collection mechanisms for distributed environments [54-58]). However, there has not been a recent survey of provenance in general ((2009)[47], (2008)[21], (2007)[48], (2005) [44, 45]). In particular, there has not been a survey, in sufficient detail, on provenance collection mechanisms. Therefore, given the current variety of provenance collection mechanisms, it seems useful that this gap be addressed in the context of scientific workflow system. In the rest of this chapter, we will survey provenance collection across a set of scientific workflow projects with an emphasis on provenance collection mechanisms.

The design dimensions of provenance collection mechanisms are presented in subsequent sections. The purpose of the current section is to build an understanding of collection mechanisms which help us designing appropriate provenance collection mechanisms for scientific workflow systems based on systems and scientists requirements. In addition, it provides a framework to compare a number of works on provenance collection works, as presented in Table 5.1 at end of this chapter. The following is the list of design dimensions of provenance collection mechanisms that are discussed in this chapter.

- Provenance collection phases in workflow lifecycle
- Workflow orientation
- Level of abstraction
- Retrospective and prospective
- Granularity of provenance information
- Accessibility of detailed information
- Model of Provenance (MoP)
- Architectural layers of provenance systems
- Coupling strategy
- Storing, accessing and querying provenance infrastructure
- Provenance representation Technique
- Type of instrumentations

These dimensions are not orthogonal and there is overlap between some aspects of some dimensions. We are going to briefly discuss each in turn.

5.1.1 Provenance collection phases in Workflow lifecycle

Provenance information can be collected during whole or a part of workflow system's lifecycle [46, 47, 176]. The provenance collection phases during workflow lifecycle contain

- *Workflow specification*: which covers all the composition, representation, data model and mapping phases in workflow lifecycle. It keeps record of all entities, relations between entities, parameters and workflow configurations before execution phase.
- *Workflow execution*: which keeps record of intermediate and output and results dataset; parameter changes; and workflow and their entities status during execution time.
- *Workflow evolution*: which records changes in a workflow specification during the lifetime of use of workflow. These changes include parameter values, changing the relationships between entities or adding/removing new entities.

5.1.2 Workflow Orientation

As explored in section 2.3, there are two broad workflow categories, scientific and business workflow systems. Business (control-flow or control-driven) workflows are designed to control connection between activities and some control-structures. A business workflow was introduced in section 2.3 as a standard model that primarily defines and employs business rules, policies and high-level management in business workflows. A business workflow represents a transfer of control from a workflow activity (process) to another activity and arranges the order of executed services in workflow systems. It represents the control dependencies and partial ordering relation of business workflow activities [44]. While, scientific (dataflow or data-driven) workflows are designed to work with data-driven applications; and consider the flow of data between/inside workflow actors. This dataflow determines the connection and interactions between activities in workflow from data producer to consumer. The collected provenance information, which is concerned with data production, is called data-oriented provenance information [44] (In this thesis, when we use the term workflow without qualification, we mean scientific workflow).

5.1.3 Level of abstraction

Most workflow systems are focused on collecting provenance information at the data and process level, while information concerning organization and knowledge also can be collected by workflow systems (shown in Figure 5.1). Provenance collection can be viewed as occurring at four levels of abstraction [19, 177], including

- *Process level:* records the order of invoked services and produced data by each service. Process provenance is similar to log systems, which record service invocations and workflow running information with corresponding time stamps.
- *Data level*: records the origins of each piece of data, including inter-process data (input, output) and even intermediate data. Data level provenance keeps track of the derivation of produced data that is inferred from process level.
- *Organizational level*: records information regarding the author and user of data, service and workflow. It could also contain some information regarding evaluation or modification of different versions of a workflow by different scientists.
- *Knowledge level*: records experiment-specific (abstract view over logging contributed by scientists) or domain-specific provenance (abstract view over the data processed during runtime contributed by scientists or data curators annotations about the data) [19, 177].

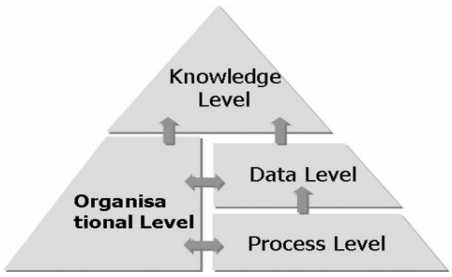


Figure 5.1.Provenance pyramid from [177].

Stevens *et al.* [177] showed the levels of abstraction in a form of pyramid as shown in Figure 5.1, revealing data provenance is derived from the process provenance. This work also shows that organizational provenance has some similarity to data and process level provenance information. The organizational metadata can be attached to provenance information in process and data level. Finally, the organizational and data level provenance information provide a high level view of provenance, including interpretations and user's abstractions [177].

5.1.4 Prospective and Retrospective provenance information

Based on the provenance collection phases, provenance information divides into prospective and retrospective [21, 26, 47]. Prospective provenance information includes the specification of workflow components; and the configuration of environments, parameters and components. Retrospective provenance information concerns data derivation and production, and is collected during a workflow instance evaluation and execution [26]. Data and process dependencies are derived from both types of provenance information, and are used to validate and reproduce the processes [21, 26]. Therefore it is important to realize which level of provenance and which level of detail is needed to record in order to satisfy the validation and reproduction process. The details and level of collected provenance information is expressed as granularity.

5.1.5 Granularity of Provenance Information

As explored in section 4.5, provenance information can be presented and collected at a variety of levels of granularity. Coarse-grained and fine-grained are two levels of provenance granularity that we consider in this section.

Coarse-grained provenance concerns the sequencing of the creation of input and output data of workflow components and processes. Fine-grained provenance considers the hierarchical structure of workflow system to represent provenance information about the interaction both within and between workflow components. It contains all information and dependencies defined in coarse-grained provenance; in addition to some additional (fine-grained) provenance information includes data produced and consumed inside processes; and computational steps applied to intermediate data (intermediate dependency) resulting in the process outputs and workflow results. In chapter 4, we defined a MoP for each level of provenance granularity that precisely representing the sort of provenance information should be collected for each level.

5.1.6 Accessibility of detailed information

Precisely representing the granularity of collected provenance information is directly related to the ability of provenance collection mechanisms in terms of accessing the workflow information. Provenance collection mechanisms might be capable of accessing different levels and types of information in scientific workflow systems including workflow, process and operating systems. The following categorizations are presented based on the level of access to scientific workflow information [21, 47, 178]

- *Workflow-level provenance*: is a level of provenance information that is captured by SWfMS. The provenance capturing mechanism can be either attached to or integrated in a SWfMS. This mechanism is usually tightly-coupled with a workflow system and enables the workflow system to capture the processes through the system's Application Programming Interface (API). Provenance capturing in this level is relatively easy, while it is not completely appropriate solution for heterogeneous environment, because it has dependency to SWfMS.
- *Activity (process or service) -level provenance*: is a level of provenance information that is captured in processes, activities or services. Its provenance collection mechanism is independent from SWfMS. In this mechanism, each process in a computational task captures its own provenance information. Therefore, the smaller provenance granularity can be captured in comparison with workflow-level and more precise information when compared with OS-level provenance information.
- *Operating System-level provenance*: is a level of provenance information that is captured within the Operating System (OS). Its provenance collection mechanism does not need to modify existing workflows and programs. It just relies on availability of OS functionality to support this mechanism. It is independent from SWfMS. It captures even smaller level of granularity by comparison to workflow-level. This mechanism captures fine-grained provenance information and needs further processes in order to reconstruct causal relationship and data-dependency.

The workflow-level provenance mechanism collects retrospective and prospective provenance, because it has access to workflow definition and it is also able to control execution. OS- and Process-level of provenance capturing mechanisms are capable of capturing retrospective provenance information. They need further actions on their collected information to reconstruct order of processes; data-dependency; and causal relationship within

and between data and processes. The workflow and process-level provenance collection mechanisms are able to adjust the granularity of provenance information, while OS-level just can collect fine-grained granularity (such as, system calls and system files accessing during task ‘s execution) [21].

5.1.7 Model of Provenance (MoP)

Recall from chapter 3, the model of provenance determines what provenance information is collected and how that information is represented. There are varieties of MoPs that are presented in chapter 4 including, OPM [49, 153] and OPMW [52]. Many of provenance collection mechanisms in scientific workflow systems (including the workflow systems presented in Table 5.1) support a form of OPM expressing for provenance challenges [49, 93, 94, 179]. In section 5.2, we explore how they represent their provenance in a format compatible with OPM.

5.1.8 Architectural layers of provenance systems

As mentioned in 2.1.3, provenance systems could be classified based on the layer in software architectural layers at which they are applied; however, there are other categorizations [45, 72, 78] for provenance system explained in that section. In our current classification, we consider Platform, Framework and Application as the choices of software architectural layers for provenance system as shown in Figure 5.2 and explained in following sections.

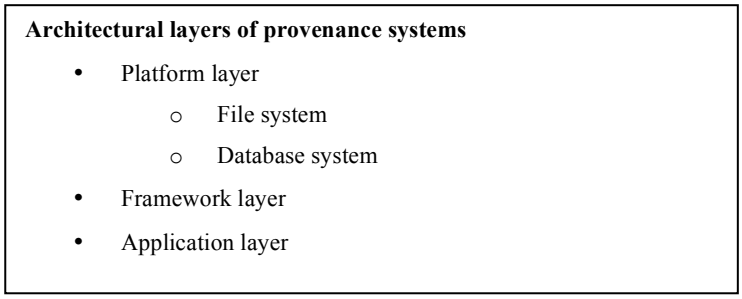


Figure 5.2. Architectural layers of provenance systems.

5.1.8.1 Platform layer

File systems and database systems work at the level of platform and almost all systems use the file systems and databases. A provenance system working on these systems can, transparently and without any intervention of applications, capture provenance information such as time stamp, fine system events and database events. Of course, only interaction of applications with file system and database are captured. This means at least some aspects of large range of application can be captured with minimum integration cost.

- *File System* is mostly used in command processing and scripting architecture. It is usually a fine-grained provenance collection approach (refer to introduction of this section), which captures the program execution provenance. Command processing systems use shell interface to have provision of created log of data transformation managed by commands. Provenance (lineage) information in command processing systems is collected from the log files and additional metadata collected and stored in a data management system during data production. A scripting architecture is useful to be used in executing the processing application, which could be another workflow application, scripting programs (such as MATLAB, Python) and invoking a service via webservice. It usually uses special library to construct log file and provenance (lineage) information for the data production [45]. These are examples of file based systems such as Lineage File System (LinFS) [180], Story Book [181], Transparent Result Caching (TREC) [85], Support for Provenance Auditing in Distributed Environments (SPADE) [168, 182, 183](refer to section 5.2.4).
- *Database systems* are very well established and systematic. The provenance systems in this approach deal with the data lineage problem: that is, when data items exist in database systems, how to determine the source data used to produced those data items. The data lineage problem is solved by an inversion technique [78] that enables generating input data from the output data. The inversion technique (explored in detail in section 5.1.11) is a relational transformation approach that is generated by user (making this inversion function is impossible sometimes) [45]. Refer to Trio [72, 86] and Panda [72, 88] in provenance in database systems categorization presented in above-mentioned categorization.

File and database data are platform entities; all belongs to one level where no intervention with applications is required. Provenance (collection mechanisms) systems in platform layer capture many aspects of behaviours. The file system is configured and works all every applications but the captured information is not necessarily the exact information that is

required. They capture very detailed and low level information when only one facet of behaviour is required and it is very difficult to reverse engineer all relevant and required behaviour of the application through such as detailed and low level information as a result of application's impact on the file or database system. For instance, further information and processing is needed to reconstruct processes ordering; data-dependency; and causal relationship in/between data and processes. The provenance systems in platform layer are capable of accessing to OS level of scientific workflow information as explored in section 5.1.6.

5.1.8.2 Framework layer

Framework or Middleware layer is featured in Service-Oriented Architectures. Service-Oriented Architecture provides a mechanism (in a middleware layer) for collecting of provenance [45, 72, 78]. This architecture does not need much cooperation with the application and it may be just sitting on top of application or workflow. Some configuration maybe required embedding applications in the framework. Many of provenance systems in Grid environments are designed in this layer such as Chimera [184], PASOA [185], Karma [186, 187] (explained in section 5.2.5) and Matrioshka [19, 55] (explained in section 5.2.2).

Framework layer provenance does not require codes to be modified. It is not restricted to specific applications. A user just needs to specify what is needed through the framework configuration. This layer also captures much more useful information for prospective provenance in comparison with the captured information from platform system level.

5.1.8.3 Application Layer

Application layer provenance embeds a provenance collection mechanism directly in the specific application of interest; as such it requires significant cooperation from the application, but it can be tailored to the needs of application. Examples of such application or domain specific provenance mechanism [78] such as GIS [83] that is the geographic domain. Applications in application layer (includes source-code and build-system [72]) collect and

manage provenance using source code control and build system such as Vesta [188] and AML [84], and it is mostly designed for a particular domain.

The Application layer captures provenance for each application and has to be rewritten for another applications. While, framework plugs in or sits on top of standard workflow and captures a narrower facet (aspect) of provenance information in comparison with the application layer, but it does not need to do anything for application at this level. Workflow systems could be placed somewhere between the application and platform layer. Workflow provides a framework that takes pre-existing code and plugs in with some effort with minimum interference with code but some configuration requires by the user. It is not quite as flexible as application layer which it could really modify code.

5.1.9 Coupling strategy

Coupling (in software engineering) is the manner and degree of interdependence between software modules [189]. We classify provenance collection mechanisms in terms of their coupling with SWfMS into tightly coupled, non-coupled and loosely coupled [47] to show how they are connected. Those coupling strategies are defined in [47] to explore the coupling of experiment data and provenance information collected from the experiment data in storage systems. In this case, in non-coupled strategy provenance information is stored in one or several provenance storages. In tight-coupling strategy, provenance information is stored along with experiment data in a same storage. Finally with a loose-coupling, provenance schema and experiment data are stored in a same storage but they are logically separated (by using mixed storage schema). Now, we introduce our definitions of coupling strategies of provenance system (collection mechanism) to SWfMS follow:

- *Tightly-coupled*: provenance collection mechanism is directly associated with either attached to, or integrated in a SWfMS (or any of its components such as workflow engine), in order to collect workflow-level provenance information. Therefore, this model is capable of collecting appropriate provenance data model in terms of retrospective, prospective, orientation and causality of provenance information.
- *Non-coupled*: provenance collection mechanism is not integrated in SWfMS; however, it may have indirect collaboration with SWfMS. In non-coupling strategy, provenance collection

mechanisms can access workflow information through the workflow storage. The non-coupled approach fits into heterogeneous environments.

- *Loosely-coupled:* provenance collection mechanism is not integrated in SWfMS while has direct collaboration with SWfMS. This model might be employed as an independent mechanism (such as, Matrioshka [19, 55] explained in section 5.2.2), in order to collect provenance information. These mechanisms also fit naturally into heterogeneous environments.

In summary, coupling strategies represent three ways that provenance system (collection mechanism) to SWfMS can connect and interact together.

5.1.10 Storing, accessing and querying provenance infrastructure

Workflow systems usually provide an infrastructure to automate the process of capturing and storing experiment data and provenance information to some extent, but in different ways. The workflow systems generally make use of a variety of mechanisms to store and represent experiment data and provenance information, such as specialized Semantic Web languages (e.g., RDF, OWL, and XML dialects that are stored as files) and relational database. Therefore, they need query languages such as SQL, Prolog, and SPARQL to retrieve appropriate results from collected information [21, 26, 44, 47].

5.1.11 Provenance representation techniques

Representation techniques are used to store and present collected provenance information to a user and the workflow system. A representation is determined based on storage architecture for provenance information, and the interface by which a user wants to see the collected provenance information. There are two techniques for representation of provenance information including annotation and inversion techniques (explained below). These techniques are selected depend on the cost of capturing and richness of provenance information.

Annotation is metadata that consists of not only the derivation history of either data or processes, but also sufficient information to reproduce the derived data and repeat the derivation process, such as the passing parameters to the derivation process and workflow version that enable data reproduction [45]. Annotations are provided automatically by application, and also are specified by a user in order to initialize parameters and configure workflow systems. User-defined annotations contains important notes and decisions and information that are not possible to capture automatically. Therefore, it is more reliable to use the automatic generated annotations by WfMS or database system [26, 44, 47].

There is some possibility to execute transformation methods on provenance data in order to recreate result data. The transformation method is used to generate the output data from the source data. If the provenance management system were capable of generating the inversion of these transformation methods, it would be possible to recreate the source data from the output data. The inversion technique (refer to database classifications of provenance systems in chapter 2) is more compact and limited than the annotation, it captures limited provenance information in a form of history of data derivation, including the process creating the derived data and identifying the source data creating the derived data [45]. It uses a transformation method to recreate result data with the help of the captured provenance information. Inversion technique also uses inverse of transformation method to recreate source data that are supplied in order to derived result data [45, 47].

The inversion technique presents compact and concise representation of derived data; however, it is sometimes not possible to provide inverse function, especially for user-defined functions [45]. The annotation technique captures not only the richer data lineage, but also parameters assigned to processes. It also keeps workflow versions in order to reproduce the result and intermediate data, it captures the semantic information and it is a pre-computed and ready to use provenance [44, 45, 47], PASOA [47, 185] has both annotation and inversion technique.

5.1.12 Types of instrumentation

Collection mechanism of provenance systems can be categorized according to the type of instrumentations (that are mechanisms to extract execution information in a real time application context) [190]. Aktas *et al.* [190] introduced instrumentation as a design decision that determines trade-off between quality of provenance information and burden on user, developer or performance of application. In [190] three types of instrumentation are proposed.

- *User annotation instrumentation*: the user annotations are a human data and description entry regarding different aspects of activities in workflow systems, which are added by user in a textual format. It can be added before and after the execution. This method imposes a low-burden on workflow and application, and a huge burden on the user. Additionally, it is an error-prone approach.
- *Scavenging instrumentation*: the scavenging instrumentation is provenance collection mechanism that captures information through existing auditing and logging tools or designated middleware mostly after finishing execution (it could be triggered after start of workflow execution. This mechanism needs post-processing to reconstruct provenance information [191, 192]. There is a chance of low fidelity [77], incompleteness and inaccurateness provenance information in this mechanism [190].
- *Source code instrumentation*: the source code provenance collection instrumentation is integrated directly into the source code of provenance collection mechanism. It could use either a mechanism that has close relation with SWfMS (tightly-coupled) or independents of SWfMS (non-coupled). This mechanism usually imposes a huge burden on the workflow designers, programmers, and also may be on the applications that run this mechanism at runtime [190], but it captures complete and consistent provenance information with no burden on users.

We categorize the instrumentation of provenance collection mechanisms according the time when they collect provenance, as explained below.

- *Post-runtime:* post-runtime (or after execution) provenance collection instrumentation is a collection mechanism that captures when is needed. It could be in two models of user annotation and scavenging instrumentation.
- *Runtime*: runtime provenance collection instrumentation is a collection mechanism that captures complete and consistent provenance information of workflows almost during all the execution of workflow systems (also may during specification phase). This instrumentation is integrated directly into the source code (Source code instrumentation model).

There are different types of instrumentation for collecting provenance information that could be applied to workflow systems based on the architecture of workflow systems. Following is a list of the instrumentations methods that instrument provenance collection mechanisms in workflow systems

- *Compiler-based Instrumentation*: It is low-level instrumentation at system level (system or function call). As used in the context of SPADE [183], instrumentation could capture provenance information from system calls (OS approach) and function calls (application approach) [171, 183]. The captured provenance with system calls approach results in process-level dependency aliasing that the function calls approach less dealing with issue. For instance, if provenance collection were in system calls approach, it would capture all files read in a data transmitted over a network connection [171]. There are other instrumentation approaches in provenance systems similar to compiler-base instrumentation called, Dynamic Instrumentation and Dynamic Taint Analysis [77] used in Data Tracker [77] to capture data provenance.
- *Event-Driven Programming* is a programming paradigm in which events determine the flow of a program. This flow of data starts by a process (or component), which is called Publisher. The publisher notifies other processes via sending an event as a particular incident occurs. Subscriber is a process that is interested in this event and receives it. In this paradigm, events are detected, and then are delivered to a handler that has subscribed to this event. This handler receives the event and takes appropriate actions. Event-Driven Programming is also known as Action-Listener, Event-Handler or Publisher-Subscriber mechanisms. Kepler [46, 144] (Action-Listener), Trident [4, 16, 98] (Publisher-Subscriber) and Karma [186, 187, 190] (Publisher-Subscriber) are a number of SWfMSs that use Event-Driven Programming for their provenance collection mechanisms.
- *Wrapper* is a design pattern to translate one interface for a class into a compatible interface. Wrapper is responsible for translating calls and transforming data from its interface into an appropriate format, which is suite for original (inner) interface [193].
    - *Wrapped Activity*: wrapper encapsulates the original activities in a component-based system. The wrapped activities are considered as black-box, which observes the input and output of the original activity. It collects provenance information from original activities and then transfers them into provenance repository [178]. Generally, wrapped activity collects course-grained granularity provenance information. It is independent from SWfMS, and enables workflow system to work on heterogeneous and distributed environments. It is possible to implement Wrapped activity with the help of “port” concept in Event-Driven Architecture and Component-Based System.

Marinho *et al.* [58] introduced a Provenance Gathering Activity mechanism that works at activity level and independently from SWfMS, that is called ProvManager. In this mechanism, each activity conceptually captures its own Provenance and transfers it to repository with the help of an adaptor activity [58]. Provenance Gathering Activity is created to collect retrospective provenance information by intercepting produced and consumed data in each activity port. After that, the original workflow and created Provenance Gathering Activity are wrapped into a new activity [58].

- *Non-functional concern mechanism* is an approach expressed in MetaObject Protocol (MOP) and Aspect-oriented programming (AOP) as explained in section 2.4. The mechanism removes the instrumentation (for example for provenance collection) from the structure of actual workflow and instrumentation is expressed as a non-functional concern. Both AOP and MOP view take perspectives that are above the functional program. They look at functional program and put a perspective on it that allows them to extract information about its operation without disturbing the way it is written or structured. In software engineering terms, it is called instrumentation expressed as a non-functional concern. We will explain the design and implementation of provenance collection instrumentation that is expressed as a non-functional concern in chapter 6 and 7.

A variety of design dimensions of provenance collection mechanisms have been presented in this section to provide a general view of provenance collection mechanism and help to design an appropriate mechanism based on system and user requirements. In the next section, a number of well-known SWfMSs and provenance collection mechanism are presented. They would be compared and evaluated in terms of presented design dimensions in section 5.3.

5.2 Review of Provenance Collection works in Workflow Systems

There are a number of approaches to collect provenance information in the context of scientific workflow system. In this section, we intend to present provenance systems including Kepler [28, 46], Matrioshka [19, 55], Provenance-Aware Storage System (PASS) [56, 57, 72], SPADE [168, 182, 183], Karma [186, 187, 194], Pegasus [18, 96], VIEW [7, 97] and Trident [4, 16, 98]. We introduced and categorized Kepler (also explored in section 3.2.1), Pegasus, VIEW and Trident as workflow-centric provenance systems in section 2.1.2.6. PASS is discussed in detail in section 2.1.2.3.

These provenance systems are capable of collecting provenance in workflow system running over distributed environments. Some of them are designed for special purposes and domains, while some of them are general purpose. We introduce each provenance system and explain how their provenance collection mechanisms operate. The way they support the OPM MoP is investigated in each work. These works are compared and evaluated (in section 5.3) based on criteria presented in the previous section. In the following paragraphs, we provide a short summary of our review of provenance systems, sufficient for a first reading; the full evaluation can be read later.

The following few paragraphs provide a summary of our survey, sufficient for a first reading of this section. Kepler, Trident and Pegasus and Karma are general-purpose workflow systems used in variety of science disciplines. Kepler's provenance recorder is capable of collecting retrospective and prospective provenance; and collecting provenance in all the workflow lifecycle. It is instrumented using an event-driven mechanism. Trident is a workflow system that builds on top of Microsoft Windows Workflow and is facilitated by asynchronous message framework (Blackboard) to construct a provenance collection mechanism. The provenance collection mechanism in Trident is instrumented with an event-driven mechanism (publish-subscribe) that is capable of collecting retrospective and prospective provenance in specification and execution phase of provenance lifecycle.

Pegasus is a general-purpose workflow systems used in various computing environments. It enables construction of high-level abstract workflows for scientists without being concerned with underlying execution environments. Pegasus efficiently maps workflows on distributed infrastructure. It is capable of collecting retrospective and prospective provenance in specification and execution phases of provenance lifecycle, similar to Trident. VIEW is an application-dependent SWfMS based on SWfMS's reference model explained in section 2.2.1 This design proposed a three-layer architecture for provenance system of VIEW.

Karma, Matrioshka, PASS and SPADE are standalone provenance systems that are capable of collecting provenance on workflow systems (but they are not workflow systems in themselves). Karma is a provenance framework instrumented with event-driven mechanism (publish-subscribe). It can be added to workflow systems in order to collect provenance data. Matrioshka is coupled with workflow systems capable of collecting provenance over

distributed environments. PASS is a provenance storage system capable of automatically recording provenance of files. SPADE is a provenance collection framework that instruments applications using a compiler-based approach and capable of collecting retrospective provenance during the execution phase of workflow lifecycle. In the following subsections, we explain the above mentioned provenance collection mechanisms, in conjunction with the extent (if any) to which they support OPM.

5.2.1 Kepler

Kepler [28, 46], as explored in section 3.2.1, is a scientific workflow system that can be used in a wide range of disciplines. It has a Provenance Recorder that is modelled as a separate concern in the Kepler system, as explained in section 3.2.2. Kepler provides an extendable framework for capturing provenance information in actor-oriented scientific workflows. It also implements several event-listener interfaces, in order to receive the provenance data. When a workflow is loaded, a provenance recorder is registered with appropriate concerns into workflow systems. When the workflow is executed, the provenance recorder processes information that is received as events, and stores it in provenance persistent storage. The provenance recorder records a variety of detailed information regarding the input and output data and their metadata; the context of workflows; the specification of entities; the intermediate produced data; and the evolution and execution information [46].

There are also other third-party approaches to the collection of provenance in Kepler. The Collection-Oriented Modelling And Design (COMAD) [152] framework is defined and used in the Kepler scientific workflow system, to support nested data collections and capture explicit data dependencies through processes at a fine-grained level of provenance information as described in detail in section 4.3.2. Another independent provenance capturing mechanism named Matrioshka [19, 55] can be coupled with Kepler to collect provenance information as explained in section 5.2.2.

In the Kepler workflow system, artifact and process are similar to the concept of data token (and collection in COMAD) and actor, respectively, thus, a process usually uses one or more tokens and generates one or more new tokens. The concept of agent [49] from is similar to the

concept of director in the context of the Kepler workflow system; however a Kepler director needs to handle some other tasks, such as ensuring adherence to the semantics of the workflow model of execution. Refer to section 4.4, the causal dependency between data token, actor and agents are shown in Figure 4.6. More specifically, the "Used" relation can be identified between process and token, in a way that in each input port of a process that uses a token (a stream of tokens or a collection). Similarly, generated token in output port of a process specify the "WasGeneratedBy" relation between process and token. The (control flow) relation of two processes is identified by "WasTriggeredBy" representing the start of a process was triggered by the completion of (predecessor) process. The "WasDerivedFrom" relation of two tokens describing a token was derived from another token that have been generated. The "WasControlledBy" relation is determined between a director and a process for each process execution.

5.2.2 Matrioshka

Da Cruz *et al.* [19, 55] presented an independent provenance capturing mechanism from workflow engine and storage, called Matrioshka. It is capable of managing provenance collection for workflow over heterogeneous cluster, grid environments, and extendable to cloud environments. Matrioshka's services are coupled with SWfMSs (Kepler [19] or VisTrails [55]) to produce provenance log for workflow systems [19].

Matrioshka's services include Provenance Broker and Provenance Eavesdrop to collect provenance and a Provenance Repository to store provenance. Provenance Broker collects and tags data produced by workflow and also listens to the produced logging stream messages (metadata) by Provenance Eavesdrop in heterogeneous environment's resources. These messages contain intermediate and final data product [19, 55]. Then Provenance Broker applies to a set of rules for experiments given by scientists regarding tags, duration of run and data transformation (which these rules considered as organization and knowledge level of provenance abstraction). Finally, the Provenance Broker dispatches the collected provenance information into a Provenance Repository. The Provenance Broker has other roles including checking for network status and supplies uniform query interface [19].

Figure 5.3 shows the provenance data schema for a provenance repository system in the form of a UML class diagram. Scientists need to set parameters and information about instances of Cloud or/and experiment host. A number of important parameters for scientists are stored in manifest file, for example, accessing virtual instances information, the number of instances to be used, the name of programs to be executed, input data and input parameters of the remote programs. The manifest file also contains a set of metadata associated with execution of workflow activities in Cloud (retrospective provenance) [55]. It should be considered that Matrioshka requires single centralized provenance storage. In addition, it assumes to reconstruct provenance based on provided log system.

The schema in Figure 5.3 is compatible with OPM, based on following correspondences,

- CloudOutput and CloudInstance represent structures in digital computing systems such as database, file, parameters, and images, so they are mapped to OPM artifact,
- CloudActivity represents a process that produced artifact or perform an action on artifacts, so it is mapped to OPM Process
- CloudUser and CloudProvider are responsible for enabling and controlling the execution of processes, so they are mapped to OPM Agent
- CloudUserWorkflow, CloudUserInstance and CloudActivityInstance are determined an agent or an artifact function in a process, so they are mapped to OPM Roles [55].

In this scheme, OPM processes, artifacts, roles and agents are represented. Therefore, the OPM dependencies can be determined during execution.

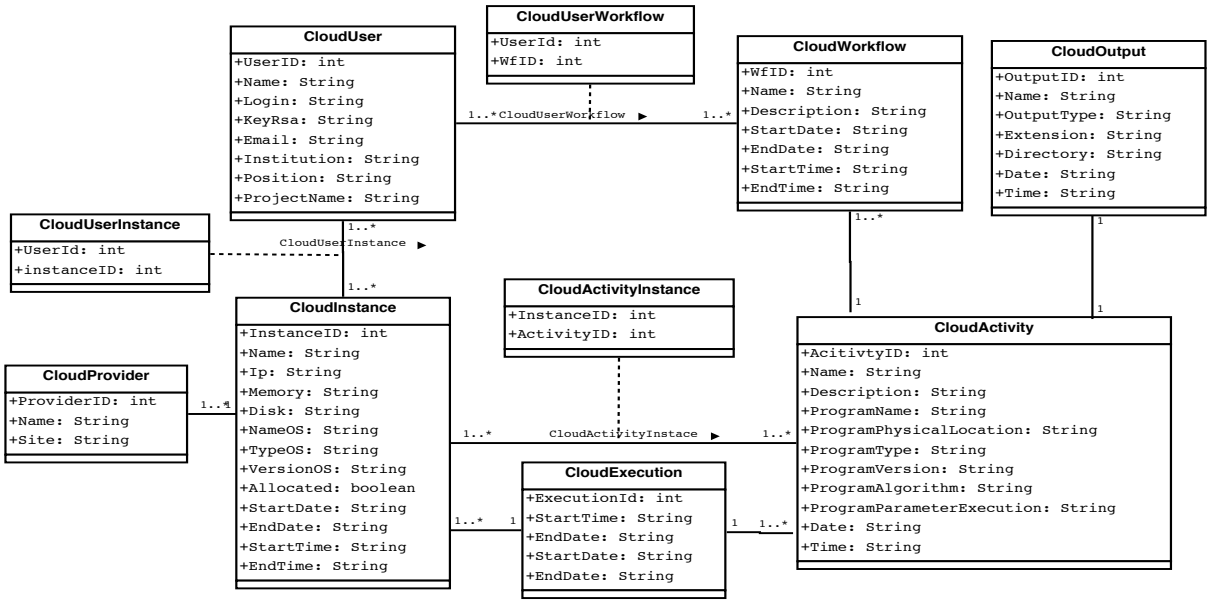


Figure 5.3. Matrioshka provenance data schema, derived from [55].

5.2.3 Provenance-Aware Storage System

Provenance-Aware Storage System (PASS) [72] is a storage system that automatically maintains and records provenance of files by observing system calls of the application that their provenance is intended to collect. The provenance records are structured in the form of attribute/value pairs such that an attribute is a unique identifier, and values could be simple values or object references. PASSv2 (a redesigned version of PASSv1) contains seven components shown in Figure 5.4 [72].

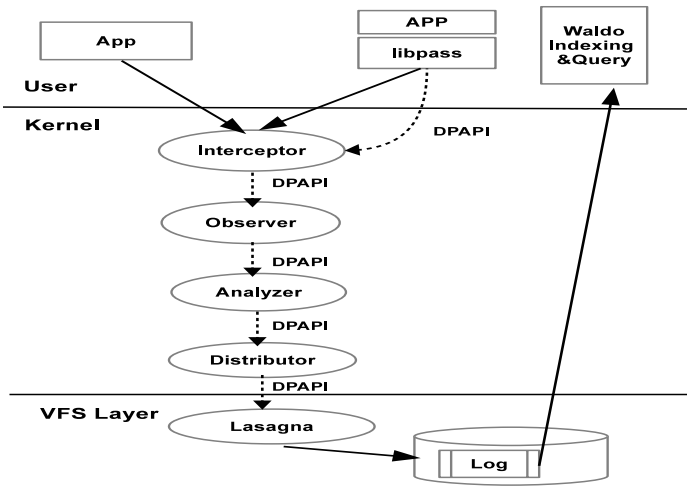


Figure 5.4. PASSv2 architecture, derived from [72]

PASSv2 has a Disclosed Provenance API (libpass in Figure 5.4) in order to send provenance information to the kernel layer of PASS. The system calls are then intercepted and passed to the Observer component by the Interceptor. The Observer generates the provenance records and process dependency based on transferred system calls from Interceptor. Analyser removes the duplication provenance and processes streams of provenance records in order to make sure there is no cyclic dependency in them. Distributor caches provenance records of kernel's object until they are stored in a persistence file system such as Lasagna [72]. Lasagna is provenance-aware file system that stores provenance records and actual data in a form of log file to ensure the consistency between them in disk. Finally, the provenance records are read by Waldo from the log file in order to store them into database (BerkeleyDB). Waldo also manages database access for query engine [72]. PASS has own query tools [72], and it uses Orbiter [195] and InProv [196] for provenance visualization.

Muniswamy-Reddy *et al.* [56, 57] presented three provenance capturing and storing architectures in the context of Cloud with the help of PASS. In these architectures, PASS automatically and transparently collects provenance information, and uses one of the architectures to store provenance information. The architectures make use of Amazon Simple Storage Service (S3), Simple DB, and Simple Queuing Service (SQS).

In the first architecture, data and provenance information is stored directly in Amazon Simple Storage Service (S3). In the second architecture, data is stored in S3 and provenance information is stored in Amazon Simple DB [56]. Finally, in the third architecture (similar to the second one) data is stored in S3 but in order to store provenance information in the Simple DB, Simple Queuing Service (SQS) is used to ensure read correctness.

PASS uses its own native provenance model (key-value format), and there is no formal translation into OPM. It has just a simple script converting PASS internal format to OPM (used in the third Provenance challenge). PASS accesses to provenance information at OS-level [21, 197].

5.2.4 SPADE

SPADE [168, 182, 183] is an infrastructure for data provenance collection and management. SPADEv1 was developed to confront a number of challenges such as provenance growth and verification latency. Due to several reasons including high overhead for experimenting new storages, handling provenance from different sources and combining additional provenance attributes, it is redesigned to SPADEv2 [168, 182, 183], which is also OPM-compliant. SPADEv2 transforms records from a domain (OS activity) into an OPM representation of provenance data. SPADEv2 utilizes a tracking module for tracking OS activities, through which it is capable of recoding dataflow dependency between files and processes. Moreover, it also records data movement across the system.

SPADEv2 provides an infrastructure and middleware for provenance recording service and reasoning about origin of data (SPADE Kernel), which makes it possible to track flow of data within a host and across hosts [168, 182]. SPADEv2 comprises components to collect,

integrate, filter, store and query data provenance in distributed environments. SPADEv2, as a decentralized model, manages provenance data on the distributed hosts where the provenance data is generated, collected, and then stored in a database in the host.

SPADEv2 has a cross-platform kernel that decouples the collection, storage, and querying of provenance metadata derived from applications, operating system, and network activity [183]. The SPADEv2's kernel is located in the operating systems; however, SPADEv2 is facilitated by compile-base instrumentation as explained in [171]. SPADEv2 is capable of automatically collecting provenance information at operating system and application level [183]. The OS provenance approach has several interfaces that are developed for different OSs to collect the OS level provenance information. While this approach would run without any modification in application level, it has some limitations, such as modelling processes as a monotonic entity which results in not capturing intra-process dataflow and not differentiating application-level dependencies [183].

5.2.5 Karma

Karma [186, 187] is a standalone provenance framework for collecting and representing provenance data. The Karma provenance collection framework can interact with workflow systems, which are driven by either user-directed workflow (no orchestration tool in use) or by a workflow orchestration system (workflow engine). Figure 5.5 represents the two-level information model of Karma, registry and execution level. The higher level of the information (registry level) contains sufficient information about services and data products to make decisions about particular data products or/and service bindings. As shown in Figure 5.5, the registry level has three entities: Abstract Method, Abstract Service (including composite service and opaque service), and Abstract Data Product. The Abstract Service could be a composite service, which represents a workflow having a relationship of "has sub-service" with one or a number of its member services, as an opaque service, which represents a black-box workflow containing unknown services. The particular sequence of instance invocations and executions are captured on the execution level, as shown in Figure 5.5. Workflow in Karma is defined in terms of services and a relationship "has next-service" which means it is followed by another service. Each Abstract Service has zero or more abstract methods that

has zero or more data product (input/output in a form of singe data or a collection of data) [186, 187].

The Karma provenance model consists of process and data provenance information. Process provenance describes the invocation of services. This invocation provides a reference for data production (input and output data). Data provenance describes the data products, and provides a reference to services invoked in producing the data products. The provenance types provide different graph structures, which make workflow and provenance traceable.

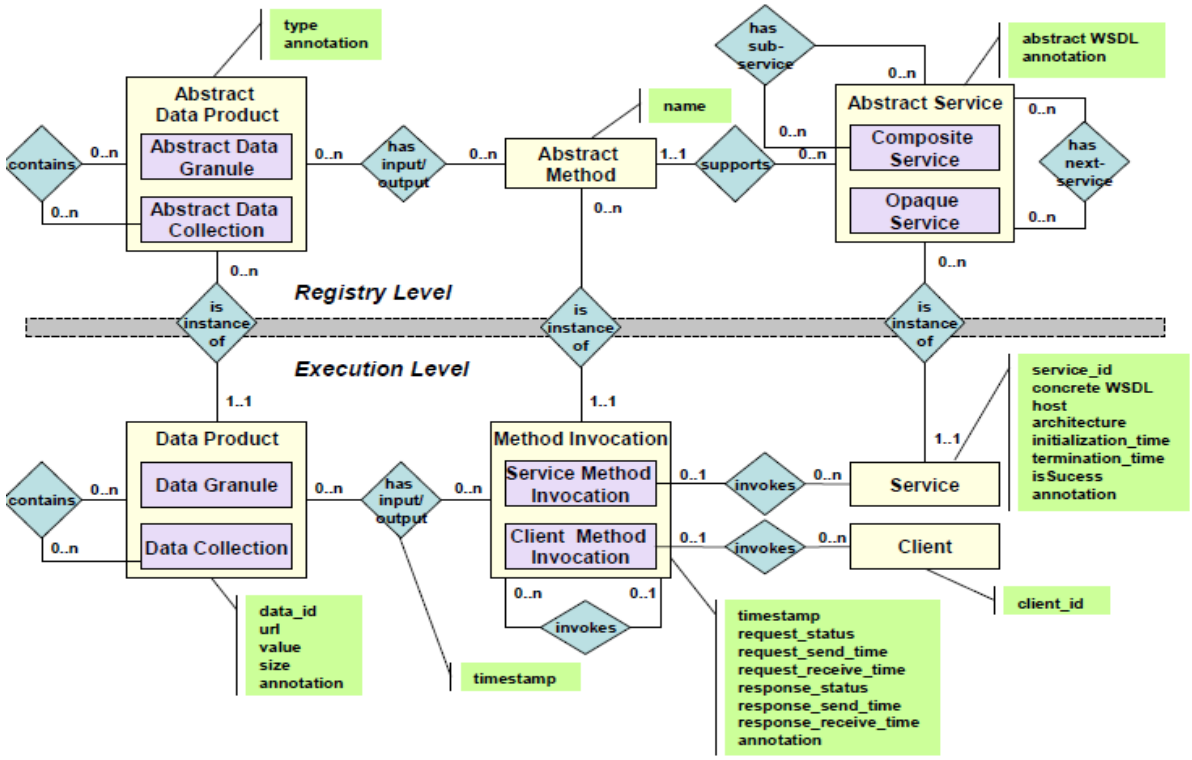


Figure 5.5. Information model composed of Registry and execution layer from [186].

The Provenance capturing layer, in Karma architecture, supports two communications forms: a publish-subscribe system and a web service model [190].

The publish-subscribe system can capture provenance in a different kind of systems, because it is not tightly coupled to WfMS presented in Figure 5.6. A publisher service generates events (or messages) and publishes it with a topic into Message bus. The subscribers listen to this message bus and can pick up appropriate events subscribed for its topic. The provenance collection mechanism tracks services that generate events and consume the events, in order to

scavenge and establish a (causal or explicit) relationship between them. This model would be compatible with the OPM model [186, 187]. Karma supports RabbitMQ and Advanced Message Queuing Protocol (AMQP), which provides scalable and portable message system with predictable and consistent latency [190]. Karma also supports Web Service [198] (Apache Axis) communication framework to transmit provenance from application to persistent storage. The provenance collection architecture of Karma based on Axis consists of three components: client, server, and Karma service. The client actor, such as a workflow orchestration engine, generates provenance events related to an entity that invokes another entity, consumes and generates data. The server generates provenance for invoked tasks and services. The Karma services with the help of handlers (Client Side Handler and Server Side Handler, which routes events to the karma server), collects Input and Output events and messages from clients and servers [190, 198].

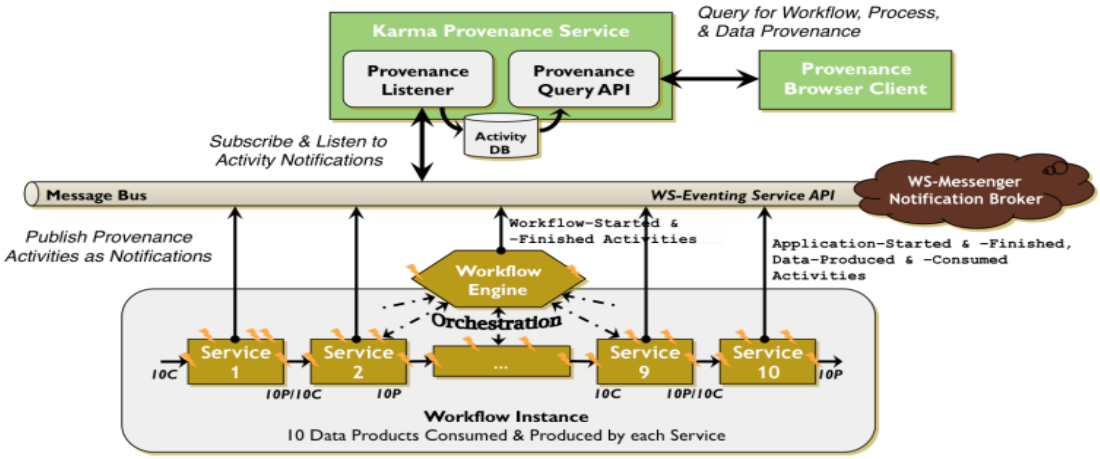


Figure 5.6. Karma publish-subscribe architecture from [194].

As explain earlier, Karma consists of three entities, Data product (single data or a collection of data), Service (compose and opaque service) and Methods (each service has zero or more abstract methods). The responsible of initiating workflow is given to a client, user or workflow engine. The Data product, Service (or Method) and Client (user or workflow engine) are mapped into artifact, process and agent in OPM, respectively [186]. Karma also has following causal dependencies corresponding to OPM,

- OPM: Used(R) relationship is determined when method invocation has single data or a collection of data as an input, represented as Method Invocation <has input>. “Method Invocation” and “has input” represent OPM:process and OPM:artifact, respectively. The Used dependency is collected for an input that is used by a method invocation.

- OPM: WasGeneratedBy(R) relationship is determined when single data or a collection of data returns as a result of method Invocation, represented as <Inverse (has output)> Method Invocation. The notation "inverse" is not define by author, but we believe it is intended to order operation in a suable way for provenance
- OPM: WasControlledBy(R) relationship is determined when a client (represented as an OPM:agent) invokes a service (represented as an OPM: process), represented as, Service Instance <Inverse (invokes)> Client.
- OPM: WasTriggeredBy () relationship is determined when an abstract service or composite service (represented as an OPM: process) in registry level has a method invocation to another abstract service, represented as

  Abstract Service <has next method> another Abstract Service, or

  Composite Service <has sub method> another Abstract Service.

  OPM: WasDerivedFrom () relationship is determined when a data item (data item 2) represented as an OPM: artifact) is resulted as a input data item (data item 1) into a method invocation (this dependency is called WasGeneratedBy() in [186]). The WasDerivedFrom dependency is represented as, data item 2 <inverse (has output)> Method Invocation <has input> data item 1.

5.2.6 Pegasus

Pegasus [18, 96] is a data and computational SWfMS deployed across different environments, such as local execution, Cluster, Grid (Condor, Nimbus, Open Science Grid, TeraGrid), Clouds (Amazon EC2). It is applied to different domains such as astronomy, bioinformatics, earthquake science, ocean science [96]. Pegasus as a workflow-mapping engine can efficiently and dynamically map and reconstruct workflow execution on distributed infrastructures. A workflow in Pegasus is created in three steps, according to [18]

- *Workflow template* is a data-independent and execution-independent abstract specification of computation that represents computational structure, components type and dataflow between data and execution.
- *Workflow instance* is an execution-independent workflow that specifies the needed input data for components and clarifies the dataflow among them.
- *Executable workflow* is a process of dynamically allocating workflow instances onto available resources in the execution environment.

Pegasus enables scientists constructing high abstract workflows without being worried about underlying execution environments or low level specifications. In this respect, Wings/Pegasus system [18] implements the first two steps (Workflow templates and instances), and Pegasus maps workflow instances into appropriate resources to execute the workflow. Therefore, Wings creates prospective provenance information (using OWL), while retrospective provenance information (using Virtual Data System) is created by Pegasus. Wings [161] is a semantic workflow system (representing a form of ontology) that is capable of mapping into OPMW ontology. The produced provenance information is stored in a relational database, and SPARQL and SQL are used for querying through provenance information [18, 21, 47]. Pegasus records provenance information regarding produced and consumed data and also the configuration and parameters of applications.

Wings assumes a number of constraints on file collection (such as, a metadata to explain how, who generates files and what is contain), component input/output, and global constraints among multiple components. Therefore Wings/Pegasus framework reasoning about application-level constraints with the help of OWL semantic presentation to validate workflow and generate application-level provenance information for new workflow data products [18]. The processes of mapping workflow instances onto available execution resources consist of a number of refinement operations (partitioned into smaller sub-workflow). Provenance information is collected for each refinement and execution step. The collected provenance is represented as a record (which contains input/output transformation), the function (that performed transformation), and optional annotation (to justify the reasoning for the performed tasks) [17, 18].

TB-Drugome project [160] uses Wings/Pegasus provenance model that fits very well into OPMW (refer to section 4.4.1). TB-Drugome project uses OPMW model where Wings considered as an abstract workflow, storing the provenance information of the workflow specification, and Pegasus stores provenance information of workflow execution.

5.2.7 VIEW

VIEW (VIsual sciEntific Workflow management system) [7, 97] is an application-dependent SWfMS that is proposed according the SWfMS's reference model by Lin *et al.* [7, 97]. VIEW is used in the field of Neurological disorder diagnosis and Biological simulations [97].

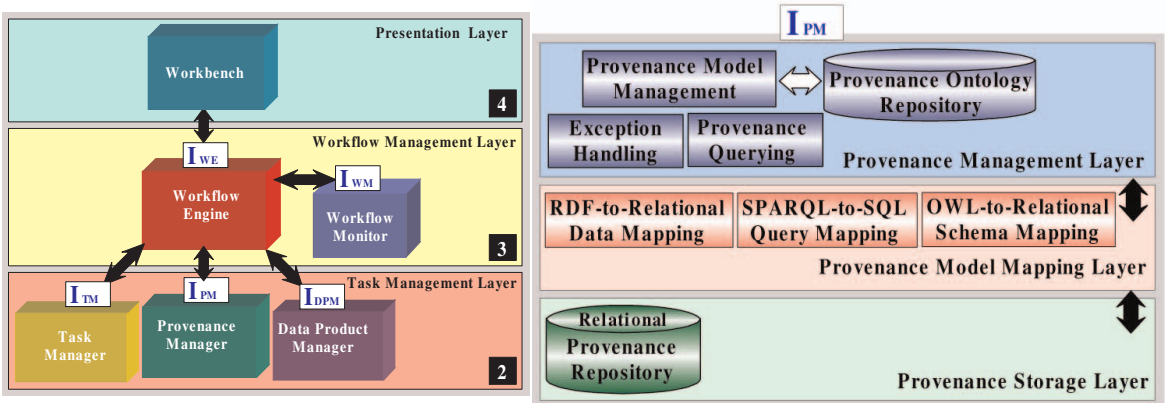


Figure 5.7. (a) Architecture of VIEW (b) VIEW Provenance Manager from [7, 97].

As shown in Figure 5.7.a, VIEW has a presentation, Workflow Management and task Management layer. A Workflow Engine in the Workflow Management Layer is responsible for scheduling, translating, control-flow managing, dataflow managing, workflow status managing and provenance collecting [7, 97]. It also has Provenance Manager Components in the Task Management Layer (as shown in Figure 5.7.a) that consists of three layers shown in Figure 5.7.b.

- The *provenance management layer* illustrates provenance from the workflow running in the form of an ontology consisting of provenance model management and provenance querying. The provenance model management deals with provenance vocabularies and domain-specific ontologies to manage the ontologies. It collects both prospective and retrospective provenance information. VIEW utilizes web ontology to meet the interoperability, extensibility, and semantic integration of SWfMS's provenance requirements. It expresses provenance information in OWL ontology and RDF ontology to serialized provenance metadata. The provenance querying component applies SPARQL queries to provided ontology [7, 97].
- The *provenance model mapping layer* is an integration layer between provenance storage layer and provenance management layer, having mapping following three mapping; 1) "OWL-to-Relational schema mapping", maps provenance metadata represented on OWL ontology to generate relational database schema; 2) "RDF-to- Relational data mapping", maps and stores provenance metadata represented on RDF to relational tuples of relational database; 3)

"SPARQL-to-SQL query mapping", transfers SPQRQL queries into SQL form in order to apply it into relational database [7, 97]. VIEW provenance collection mechanism develops RDFProv [199] and OPMProv [164] systems to storing and querying purposes of provenance information. RDFProv is a provenance-model agnostic that is capable of managing different captured model of provenance based on provided ontology representing concepts and relationships in the model. RDFProv is based on RDF and capable of serializing and querying provenance information with the help of SPARQL [164]. OPMProv expresses the OPM model of provenance information in relational database (entity-relation model) [164].

- The *provenance Storage Layer* includes relational DBMS, which is queried by SQL [7, 97].

5.2.8 Trident

Trident [4, 16, 98] is a scientific workflow for constructing and running scientific data analysis workflows. Trident is a data driven scientific workflow built on top of the "Windows Workflow" foundation for business workflow. Trident was firstly introduced in the field of oceanography (NEPTUNE) [1, 4, 35], and astronomy (Pan-STARRS) [1, 4]. Trident is facilitated by runtime service including provenance, fault tolerance, monitoring and scheduling service [98, 200].

Provenance information in Trident includes static information collected from the composition stage of workflow in the workflow composer, while other, dynamic, information is collected during runtime. The dynamic information consists of tracking the submission of instances in workflow, and tracking the events of workflow execution that are collected in the execution service and workflow engine respectively. The data-entity relationship of static and dynamic information is presented in Figure 5.8. The static information is the abstraction of the workflow with parameters and data values before the actual execution (blue entities in Figure 5.8). It contains information and signature of activities that are imported into the workbench (in Activity and Activity Parameter entities); and the new composed data and control flows activities information (in Activity Sequence and Parameter Assignment entities) [16]. The dynamic information (green entities in Figure 5.8) represents the job submission, execution time of activities and workflow, parameters and data values came in/out of instances and execution status [16].

Trident uses the Blackboard (an asynchronous messaging framework) mechanism to handle the events and messages. The flow of control is tracked by an event handler in Trident, while flow of data (the input/output data in activities) are captured and considered as a customized user events [16]. Blackboard services (shown in Figure 5.9) that are based on the publish-subscribe architecture. The Blackboard services are implemented base on the Windows Communication Foundation (WCF) in order to provide communication between Trident components and to hide all network communication behind WCF interfaces implementation of publish-subscribe architecture (utilized by Decentralized Software Services (DSS)) [201].

In this architecture, publishers send generated messages with a topic to blackboard, following that they are passed to subscribers that subscribe for the topic. The provenance service generally subscribes to data, which is published by other components such as execution service and Windows Workflow Engine. The Blackboard serves runtime tracking and logging services as well, to track information across distributed computational nodes, to provide a pluggable interface for messages in workflow. The Blackboard publish-subscribe model utilizes with fault resilient message delivery from message creator to subscriber. Moreover, this model enables delivering a message to several subscribers [16].

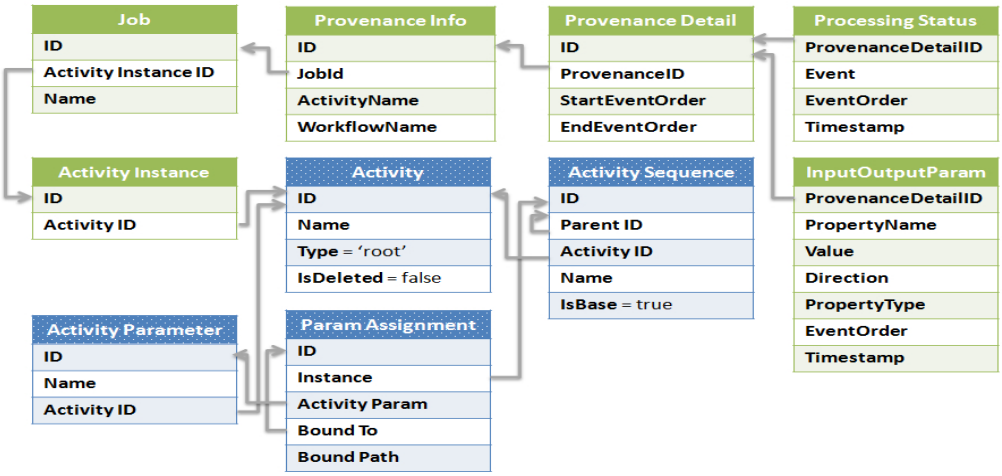


Figure 5.8. Provenance Data Model in Trident from [16].

Trident stores the captured provenance information and workflow metadata in the Trident registry. The Trident registry has a data access layer in order to provide an abstraction of data storages, which are used by the workflow. Trident data storage collaborates with a variety of types of storage from local to cloud base storage such as, Amazon Simple Storage Service

(S3), SQL Server Data Services (SSDS) or SQL server database [98, 200]. The registry is configurable with an XML schema file. Trident's data model is completely compatible with OPM for purposes of interoperability. The registry supports .NET Language Independent Query (LINQ) mechanism to generate the queries through the stored data in registry [16].

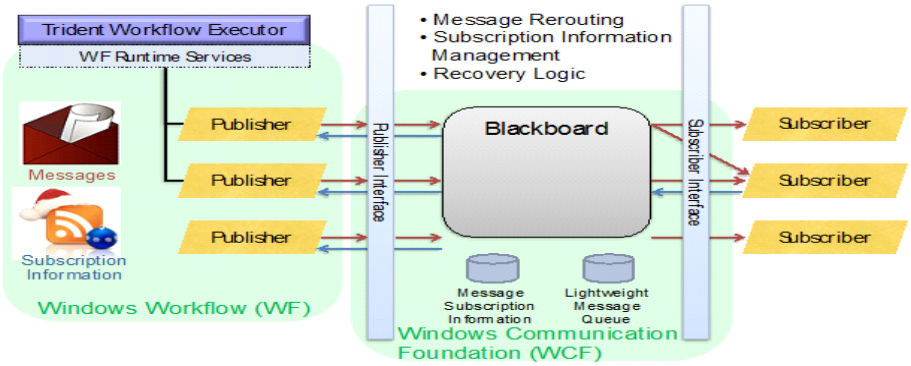


Figure 5.9. Blackboard architecture from [202].

5.3 Comparison and Discussion

There are several approaches to collecting provenance information in the context of scientific workflow systems, which result in a variety of design dimensions for provenance collection mechanisms in workflow systems. A number of provenance collection mechanisms were presented in the previous section. We intend to compare and evaluate mechanisms of collecting provenance in the workflow systems as shown in Table 5.1 based on the design dimensions presented in this chapter.

The purpose of this Table is to compare the works analysed in the preceding section, along significant design dimensions, in order to provide a greater understanding of existing provenance collection mechanisms. In subsequent chapters (6 and 7), we build on this understanding to put forward new provenance collection mechanisms. These will be evaluated based on the design dimensions presented in this chapter.

In the workflow systems categorization discussed in the section 5.1.2, workflow systems could be scientific or/and business workflow systems. Kepler, Pegasus, Trident and View are in the scientific workflow system categorization; however, Trident and VIEW utilise some control components. Trident is based on Windows Workflow Foundation, which contains

control activities using the sequential or flowchart style of running. It causes Trident to potentially be facilitated by control activities of Windows Workflow Foundation. It should be noted that PASS, Karma, SPADE and Matrioshka are not a workflow system; they are software infrastructure that supports mechanisms of provenance collection in various ways.

Table 5.1. Summary of design dimensions of surveyed provenance collection mechanisms.

	Kepler	Matrioshka	PASS	SPADE	Karma	Pegasus	Trident	VIEW
Provenance collection phases (SP) Specification (EX) Execution (EV) Evolution.	SP, EX, EV	EX	EX	EX	EX	SP, EX	SP, EX	EX
Workflow Orientation	Scientific	NA	NA	NA	NA	Scientific	Scientific	Scientific
Level of abstraction	Data &Process. Organizational	Data,	Data	Data	Data &Process	Data &Process organizational	Data &Process.	Data &Process.
Prospective (P) Retrospective (R)	P: MoML R: MoML	P: N/A or XML manifest (use other SWfMS) R: Relational	P: N/A R: Relational & XML	P: N/A R: Relational	P: BPEL R: XML	P: OWL R: Relational	P: Relational& XML R: Relational & XML	P: N/A R: Relational & XML
Granularity	Fine	Fine	Fine	Fine	Fine	Fine	Fine	Fine
Accessibility of detailed information	Workflow	Workflow	O.S	O.S	Workflow	Workflow	Workflow	Workflow
MoP	OPM	OPM	OPM	OPM	OPM	OPM	OPM	OPM
Architectural Layer	Framework	Framework	Platform	Platform	Framework	Framework	Framework	Framework
Coupling strategy	Tightly-coupled	Loosely - coupled	No-coupled	No-coupled	Loosely - coupled	Tightly-coupled	Loosely - coupled	Tightly-coupled
Storage	RDBMS,XML, Files	RDBMS	Berkeley DB, EC2, S3, SimpleDB	Berkeley DB & graph database Neo4J & H2	XML RDBMS	RDBMS	RDBMS, XML, Amazon S3, SSDS	OWL & RDBMS
Query support	Prolog supported SQL Query inside source code Java API	SQL& Browse API	Proprietary query Tool PQL (Path Query Language) Orbiter and InProv.	SQL based querry tools with some native functions expressing graph & path	Karma query API, SQL	SPARQL	Language Independent Query (LINQ)	SPARQL SQL
Type of instrumentations	Runtime	Post-runtime	Runtime	Runtime	Runtime	Runtime	Runtime	Runtime
	Source-code	Scavenge	Source-code	Source-code	Source-code	Source-code	Source-code	Source-code
	EDP	EDP	--	Compiler-based & application level	EDP	--	EDP	--

Section 5.1.1, explores the provenance collection phases in workflow lifecycle. All discussed scientific workflow systems discussed in this chapter, collect provenance information at least

during the execution phase of workflow lifecycle, while Pegasus and Trident collect provenance information during specification phase, as well. The specification phase in Trident is called static information. Pegasus workflow system uses Wings as an abstract workflow, which contains prospective information that is directly related to specification phase. Kepler could collect provenance during all phases. Similarly Matrioshka by itself can collect provenance during execution, but it is coupled with other SWfMS like Kepler or VisTrails, in which provenance in specification and evolution phases could be collected by a coupled SWfMS. We can argue that all collection mechanisms that do not support a specification phase, collect limited prospective information. In Table 5.1, Karma does not have specification phases though it can be argued because it has a registry level containing a sufficient level of information about services and data products to make decisions about particular data products or/and services binding.

All scientific workflow systems (presented in this chapter) collect at the data level of provenance (explained in section 5.1.3) because the origin of data is important for them. Kepler, Karma, Trident and VIEW also collect process level of provenance information to record the order of services invoked in producing data. Pegasus in specification (with the help of using Wings), and Kepler in both specification and evolution phases, collect the organizational level of provenance as well; however it would be fair to say all workflow systems that collect data and process level of provenance may have a sort of organizational level of provenance to some extent (having limited information regarding organizational level, such creator and launcher of workflow), but it does not mean that they are considered as organizational. Karma collects data level of provenance in a form of messages - it analyses messages to generate the process level of provenance information.

According to the definition presented for prospective and retrospective provenance information in section 5.1.4, there is a relation between workflow lifecycle and prospective and retrospective provenance information, such that the retrospective provenance information is collected for all the scientific workflow systems that at least collect provenance during execution phases of workflow lifecycle. The collected retrospective provenance information (in presented workflow systems in Table 5.1) is mostly stored in relational databases, while in Trident, PASS, Karma and VIEW it can be stored in Markup language (XML) format. It appears that Kepler stores its prospective and retrospective information in a Modelling

Markup Language in XML (MoML) [21, 46]. Pegasus uses the OWL format for its prospective provenance information, while Karma uses aspects of BPEL for prospective information in high level information regarding services and data products.

As mentioned in section 5.1.6, the granularity of collected provenance information is related to the accessibility of the workflow information. The collected provenance information in the workflow systems, mentioned in this chapter, implement fine-grained granularity, however they are not all at the same level of detail and abstraction. PASS and SPADE capture some of the very fine-grained and system level information because they have access to workflow information at the operating system level, while the rest of provenance collection mechanisms have access to provenance information at the workflow level of information. Even though, there are different level of details and types of collected provenance information for those provenance collection mechanisms accessing at workflow level of information, such as Kepler can also collect intermediate data produced inside processes, Kepler (with the COMAD director) collects provenance at the level of stream of tokens; and Pegasus collects provenance at the level of files and nested file collection. The fine-grained provenance systems mentioned in this section do not necessarily collect fine-grained provenance systems at the same level that we explored in section 4.5.1.3.

All of the workflow systems in Table 5.1 derived a form of expression using OPM for provenance challenge.

For architectural layers of provenance systems, except for PASS and SPADE (as previously noted, PASS is storage systems, and SPADE is infrastructure), the provenance collection mechanisms presented in chapter 4 are in framework layer of software architecture. SPADE uses the platform layer (file system level) of software architecture for collecting provenance information. PASS similarly uses the platform layer but it could be applied to file system and database level. The rest of workflow systems may be firstly designed to use in certain domain (such as Trident in oceanography [1, 4, 35], and astronomy [1, 4]), but because most of them (for instance, Karma, Trident, Kepler, Pegasus, and VIEW) have a framework that makes them usable in different domains, so they are considered in framework layer of software architecture.

The coupling strategy of provenance collection mechanism to SWfMS is discussed in section 5.1.9. Provenance collection mechanism in Kepler, Pegasus and VIEW are integrated into their SWfMS, so they are tightly-coupled with SWfMS. Trident and Karma using the publisher/subscriber mechanism that needs to have indirect collaboration to a SWfMS, so they are loosely-coupled. Similarly, Matrioshka is loosely-coupled with SWfMS, although it is attached with SWfMS to collect the provenance in workflows running over distributed environments. PASS and SPADE are more concerned with storage systems and have no-coupling with SWfMS.

Nearly all of the presented workflow systems in Table 5.1 support Relational Database System and Markup language to store their provenance information. PASS, in addition of Berkley DB, can use Cloud system storage such as EC2, S3 and Simple DB. Trident uses wide range of storage namely amazon Simple Storage Service (S3), SQL Server Data Services (SSDS) and SQL server database. Trident supports querying provenance information by Language Independent Query (LINQ). Pegasus supports SPARQL and Karma supports SQL for their querying, while VIEW supports both of SPARQL and SQL. Kepler uses prolog for querying while it is possible to have SQL Query inside its source code Java API. Matrioshka, PASS, SPADE using an API to making querying more user-friendly for users in different way.

As discussed in section 5.1.11, it seems that all the above-mentioned workflow systems use an annotation technique for representing their collected provenance information to provide a richer set of provenance information that the annotation technique is capable of presenting.

The time and type of instrumentation of provenance collection mechanisms (introduced in section 5.1.12) have significant influences on granularity and quality of collected provenance information; it also has a huge impact on performance of workflow systems (particularly in runtime execution). Matrioshka and Karma are facilitated by scavenging instrumentation to collect provenance information from existing logging system, so they are post-runtime. The rest of approaches are at runtime and they use source-code provenance collection instrumentation; however some of them (like Kepler and VIEW) using user annotation, as well. SPADE is instrumented in two approaches in OS and application level. OS level is compiler-base instrumentation. Karma, Matrioshka and Trident are Event-Driven

Programming with a publish-subscribe architecture and Kepler is Event-Driven Programming (EDP) with event-listener architecture. For other systems, such as Pegasus, VIEW and PASS, we do not have access to sufficiently detailed publication information to accurately characterize the method of instrumentation.

5.4 Summary

In this chapter, a survey of provenance collection mechanisms in scientific workflow systems is presented to explore how provenance mechanism works and how their design attributes would influence of provenance systems. The design dimensions of provenance collection mechanisms in the context of scientific workflow systems are defined and then examined through a number of projects (presented in section 5.2). The evaluation of the provenance collection mechanisms are presented in Table 5.1 and explained in the section 5.3. This chapter provides a sufficient level of understanding and background for us in order to design our adaptive provenance collection mechanisms presented in chapter 6 and implemented in chapter 7.

6 AN ADAPTIVE PROVENANCE ARCHITECTURE IN SCIENTIFIC WORKFLOW

We explored and surveyed provenance collection mechanisms in chapter 5 and Model of Provenance in chapter 4, in order to provide understanding and background of how provenance mechanism works and how its design attributes would influence of provenance systems in workflow systems. We will use the design dimensions introduced in chapter 4 in our provenance collection mechanisms in this chapter.

In this chapter, we design our adaptive provenance collection mechanisms that employs separation of concerns in an adaptive provenance architecture, separating functional and behavioural concerns. As explored in section 2.4, functional concerns are expressed in the base-level application (such as scientific workflow systems) and the behavioural concerns in our thesis are expressed in two ways: reflection (such as MetaObject Protocol (MOP)) [60-68] and Aspect-Oriented Programming (AOP) [59, 60, 68-70]. There are many intermingled concerns in each application of such workflow systems. We aim to separate and untangle the concerns related to adaptive provenance collection mechanisms.

We focus on the use of adaptation in provenance collection mechanisms and demonstrate the design viability of our new provenance design in a workflow architecture, in this chapter. We use MOP and AOP software techniques to express the separation of concerns in our design. The design and development of our adaptive provenance architecture untangles the adaptive-granularity and provenance-collection concerns, so that we can more easily offer adaptive provenance collection mechanisms.

In section 6.1, we provide a number of scenarios motivating the benefits of having adaptive provenance collection mechanisms capable of adjusting provenance granularity. We will explore the design dimensions of these mechanisms to provide a clear understanding of desirable properties that our designs of adaptive provenance collection mechanisms are going to present.

In section 6.2, our design for workflow architecture is motivated by features offered in the workflow controller. The workflow controller enables the adaptive provenance collection mechanism to communicate with the environment that the workflow is embedded in and

informs the collection mechanism about the current status of resources and requirements. We are trying to show provenance systems are improved by constructing and factoring aspects of provenance design in the workflow architecture. We demonstrate a clear provenance design in workflow architecture that has a distinct and separate provenance architecture. We show the viability of this design, at least in principle, as useful provenance systems that can be further designed and developed using ideas and techniques in a case study shown in chapter 7.

In this section, we design two (MOP and AOP) architectures for provenance systems that adaptively collect provenance at different levels of granularity. These architectures have desirable properties, such as adaptability and applicability. The designed provenance system in these architectures can be viewed separately (from other components in the workflow architectures), while also having a clearly identified means of integration with the host workflow system. This separation of provenance from workflow architecture is a key point of our adaptive design techniques.

Section 6.3 provides a number of design properties for adaptive provenance architecture concerning scientific workflow systems. A MetaObject Protocol (MOP) is proposed for adaptive provenance architecture that has provenance-collection, distribution and adaptive-granularity meta-behaviours. A design for each meta-behaviour is explored in in the context of the Enigma meta-level application introduced in section 2.4.1.6. AOP is another adaptive software technique that we use to design and implement our adaptive provenance architecture; it is presented on section 6.4. Our MOP and AOP mechanisms are implemented in two case studies presented in the next chapter.

6.1 Adaptive provenance in scientific workflow systems

As explained in chapter 4, changes in execution models affect the way provenance collection mechanisms work. Many workflow systems in different disciplines need to run in distributed environments [203]. Therefore, provenance collection mechanisms in distributed environments face new challenges [47, 178, 204] that require novel design and development to improve the efficiency and feasibility of the provenance collection mechanism in a distributed context. The design of provenance collection mechanisms determine how they efficiently and dynamically adapt and respond to environmental changes. In this work,

environmental changes refer to changes in the underlying infrastructure of the system. In this chapter, we focus on adaptability of provenance collection mechanisms in provenance system applied to scientific workflow systems.

We will explore illustrative scenarios in workflow systems as shown in Figure 6.1. Figure 6.1 shows examples of how workflow activities in workflow systems might run over different computing environments, network characteristics and policies. Workflows can be run on resources on the local host (presented in workflow activities 3 and 5 in Figure 6.1) or remote hosts including Cloud, Cluster and Grid infrastructure (presented in Figure 6.1 in workflow activities 1, 2 and 4, respectively). They also may run partly on resources in the local host and partly on resources in remote hosts. The workflow activities might be deployed over networks with a variety of characteristics used in the infrastructure of computing environments [55, 58], as shown in Figure 6.1 (thicker arrows represent network with better network characteristics such as low latency and high bandwidth similar to connection from workflow activity 1 and Cloud environment). We consider how workflow structure and deployment of components might change in response to changes in environment.

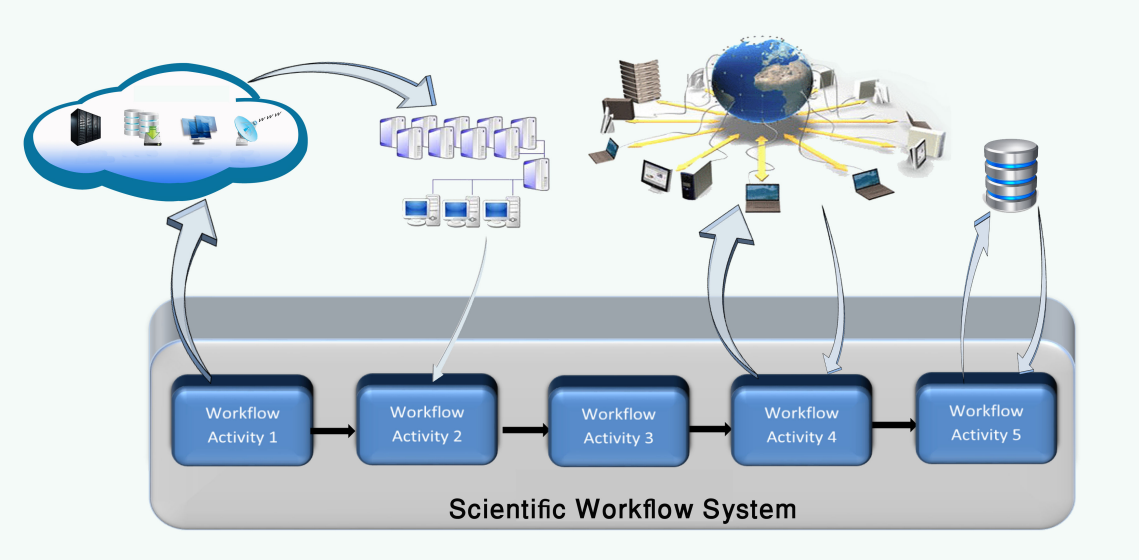


Figure 6.1. Workflow system.

In Figure 6.1, the interaction between workflow activities during execution are coordinated by both orchestration [100] and choreography [100] approaches, explained in section 2.3. The orchestration approach is applied to the workflow activities 4 and 5 while the workflow activities 1 and 2 are coordinated by choreography approach where the output results of the

workflow activity 1 are directly transferred as input data to the workflow activity 2 without being transferred via the workflow system.

Workflow systems can often run be run over such heterogeneous environments with different computing environment, network characteristics and policies while they have fluctuation in availability of resources, characteristics of networks and changing of policies. Consequently, provenance collection mechanisms in these environments should be able to adapt to these changes. For example, restrictions on computing capacity or lower network capacity limit the ability of provenance system to collect, transfer, and store provenance information. Therefore, provenance collection mechanisms should be adapted to each situation, such as collecting and sending less provenance information.

A provenance system is designed to collect, manage and store provenance information. The provenance system could be considered as a service that is initiated by workflow system. This service takes environmental parameters as input parameters, and adapts its provenance collection model based on these input parameters. Following that the workflow system returns the workflow result and provenance information. The changes in the environmental parameters motivate having an adaptation mechanism in the provenance system, as explored in the rest of this section.

The trigger of adaptation of the provenance system is based on certain criteria that need to be defined on environmental parameters passed as inputs to the provenance service (for Information Service shown in Figure 6.2 and 6.3). The criteria that trigger adaptation are usually related to system performance, the cost of collection, the size of collected provenance, network situation, and network reconfiguration. Some example of adaptation scenarios follow

- Scenario 1: This scenario is concerned with transferring provenance information to provenance storage. Some workflow activities (including such as workflow activities 1, 2 and 4) in Figure 6.1 are running in distributed environments such as (Grid, Cloud and Cluster). The collected provenance information in these environments might be transferred to provenance storage in the local host. In other situations, some workflow activities (such as workflow activity 5 in Figure 6.1) are running on the local host but they may require transfer of collected provenance information to a remote host in distributed data storage as shown in Figure 6.1.

In both situations mentioned, the collected provenance information might be transferred to another host through a network that may have some constraints, such as long distances, low bandwidth, and high-latency. Moreover, the size of produced provenance information in the workflow system could be more than the size of workflow input and result data. In this situation, large amounts of provenance information may need to be transferred through the network (with possible constraints) to be stored in data storage. Therefore, the provenance system or provenance collection mechanism can benefit from ability to adapt to unanticipated changes in network characteristics. For example, the collection mechanism can collect less provenance information by changing the provenance granularity from fine-grained to coarse-grained, in the presence of adverse network limitations.

- Scenario 2: This scenario is concerned with the status of resources and user requirements. Performance of system (such as load balancing in computing and storage resources inside workflow executed in the computing environment), resource availability, price of resources, user budget, and cost of provenance collection (the cost of achieving a certain or various levels of granularity) are constraints in computing environment that may trigger adaptation. The adaptations might include the adaptation in the level of provenance granularity. The provenance collection mechanism could be adjusted to collect provenance in different level of provenance granularity depend on some of the mentioned criteria. For example, in workflow activity 1 and Cloud environment in Figure 6.1, the system performance is in good condition, which means computing resources are available, data storages have enough space and network is in good condition (high bandwidth). The user can request for fine-grained provenance and it can be handled. When the system performance is not in good condition (for example the workflow activity 3 is not high-performance computing), the user request of fine-grained provenance may not be able to be handled. Therefore, the provenance granularity would be adjusted to course-grained provenance based on the system.

  As mentioned in our scenario in section 1.3, price is another concept in Cloud and Grid computing environment (refer to [34] for market-bases or economy based Grid model) that effects user requests for a specific level of granularity. The price of resources varies based on demand for resources in Cloud and Grid environment [34, 205], and it is even possible that the price of resources at the time of resource allocation may be greater than the assigned budget for workflow run. Even in case of having enough system performance, the budget is a constraint (because the user assign a specific budget for the experiment (job) run). Therefore the workflow collects less provenance information to keep on budget.

- Scenario 3: the migration of tasks in workflow activity is another type of reconfiguration that workflow system may deal with. For example, when a workflow run in one host then it gets saturated. The workflow system is reconfigured to identify another node (workflow activity) that has high-intensive computing, and then migrate it to another host. For example in Figure 6.1, the workflow activity 4 is not powerful computing resource and stores provenance information in the local host. The workflow system presented in Figure 6.1 is reconfigured and tasks in workflow activity 4 are migrated to the remote host and executed on a Grid computing platform (as shown in the Figure 6.1). The Grid computing platform has powerful computing resources while it also has to transfer the collected provenance data to local host (where the workflow activity is located) through low bandwidth network channels from Grid computing host to local host (narrow arrow from Grid computing host to local host show a network channel with low bandwidth, high latency or long distance).

  An adaptation in network topology is a type of adaptation whereby a task is migrated to another node or host. If provenance is being collected at the same time of task migration, then that application level adaptation will have implications of provenance. It is possible to see at the nature of application that it has high computation to communication ratio and not much data communication out of the network, but complex calculation that internally generates lots of provenance information (that it has high communication to computation ratio). Therefore, in this case for provenance point of view, it is unfavourable to reconfigure the network. Consequently, the adaptation of computation topology needs to be handled alongside of provenance topology adaptation to satisfy the requirement from the both computation and provenance aspects. In this scenario, the requirement for successfully doing the computation is different from the requirement for collection of provenance at the level we normally like to collect. Thus the provenance collection collects provenance at lower granularity to meet the requirement of computation aspect (migration to a host with high computation to communication ratio). It collects less data that needs to be sent back to where the provenance data are requested.

In this section, the necessity of adaptation in provenance systems in the context of scientific workflow was explored through some scenarios. In the next section, we explore the necessity of having an adaptive provenance collection mechanism to adjust provenance granularity.

6.1.1 Adaptive Provenance Collection Mechanisms

As explained in scenario 1, workflow systems can possibly deploy over networks with a member of characteristics that might be changed during the execution of workflow system. These changes will effect the mechanism of provenance collection. In this situation, if collected provenance information needs to be transferred, the amount of captured provenance information should be adjusted according to the underlying infrastructure and network situation as indicated in scenario 1. The changes in the amount of captured provenance information are managed through adjusting the level of provenance granularity that is collected by provenance collection mechanism.

Scenario 2 and 3 also manage the changes in computing environment by changing the level of provenance granularity. In addition, a workflow would possibly run on distributed environments with a number of unreliable and unpredictable resources, which their availability and performance may change frequently (environmental changes). Thus, a provenance collection mechanism requires capturing sufficient level of provenance according to the availability and performance of resources. Therefore, existence of an adaptive provenance collection mechanism in order to capture different granularity of provenance is warranted to satisfy several types of changes in environmental situations, user requests and also SWfMS requirements (such as rerun-ability, reproducibility and appropriate provenance information), explained in scenario 1, 2 and 3.

An adaptive provenance collection mechanism, capable of adjusting the level of granularity, can scale back in provenance granularity (for example from fine- to coarse-grained provenance) when facing lack of resources or other constraints (as mentioned in scenario 1 and 2); or scale up if it is requested and there is no constraint. Therefore, each level of granularity needs to know which sort of provenance information should be collected. As mentioned in section 4.5, a Model of Provenance (MoP) can represent each level of granularity. The connection between the concept of MoP and the concept of provenance granularity was discussed in section 4.5, and three coarse-, medium- and fine-grained MoP, as examples were illustrated in detail in that section. In this thesis, we will represent collected provenance information in a format compatible with the OPM MoP (see section 4.4 for detailed information about OPM) in a multiple-granularity MoP as described in section 4.5.2.

Two implementations of adaptive provenance collection mechanisms are presented in multiple-granularity MoP. They are compatible with OPM MoP shown in chapter 7. In the next section, we provide some expected and desired features for the design and implementation of adaptive provenance collection mechanisms. We use design dimensions that are defined in chapter 5 for provenance collection mechanisms.

6.1.2 Desirable design dimensions for adaptive provenance collection mechanisms

The design dimensions presented in chapter 5 provide sufficient understanding of the possible properties of a collection mechanism. A collection mechanism can be designed based on system and scientist requirements. In this section, the desirable and expected design dimensions of our adaptive provenance collection mechanism are presented to clarify its design principles.

Table 6.1. Desirable deign dimensions.

Workflow Orientation	**Provenance collection phases**	**Level of abstraction**	**Prospective (P) Retrospective (R)**	**Granularity**	**MoP**	**Coupling strategy**	**Type of Instrumentation**	**Storage**
Scientific	Specification and Execution	Data &Process	P: XML R: XML	Multiple-granularity	OPM	Loosely-coupled	Non-functional concerns	XML

As mentioned in Table 6.1, we aim to design a provenance collection mechanism that is applied to scientific workflow system which collects provenance during the specification and execution phase of the scientific workflow lifecycle. It should be capable of collecting both the data and process level of provenance information abstraction; however, it is mostly concerned with data level.

The desirable provenance collection mechanism should be loosely-coupled (explored in section 5.1.9) to the workflow system in order to directly collaborate with SWfMS whilst not being fully integrated in it. In addition, the loosely-coupled workflow system makes our system fit to run on distributed and heterogeneous environments.

The provenance collection mechanism is designed to collect multiple-granularity of provenance information and conforms to the OPM structure; therefore, storing provenance in XML file format.

The adaptability and multiple-granularity provenance features of a provenance collection mechanism have a direct relationship to a number of design dimensions such as accessibility of detailed information (explained in section 5.1.6), architecture layers of provenance systems (refer to section 5.1.8) and type of instrumentation (refer to section 5.1.12). We investigate adaptability and multiple-granularity provenance features of a provenance collection mechanism along the following design dimensions.

1. Accessibility of detailed information

The granularity of collected provenance information is directly related to the ability of provenance collection mechanisms in terms of accessing the information of the underlying workflow system architecture. This mechanism may be able to access different levels of information in scientific workflow systems including workflow, activity (process) and operating system. As mentioned in section 5.1.6, the OS- and activity-level of provenance capturing mechanisms are able to collect relatively fine-grained information in comparison with workflow level; however, they just capture retrospective provenance and need further processing to reconstruct causal relationship and data-dependency (we are going to collect and store retrospective and prospective provenance in OPM-compliant format thus we need to store this information with their causal relationships). The workflow-level can collect both retrospective and prospective provenance, because it has access to the workflow definition and information of workflow execution. However, it cannot capture the fine-grained information that is important for this thesis. We are aiming to design an adaptive provenance collection mechanism that is capable of collecting provenance information at multiple-granularity levels. None of the workflow, activities and operating levels do not completely satisfy the requirements of collecting multiple-granularity provenance information.

It is desirable for a provenance collection mechanism to be able to access all the levels of information, thus enabling the capturing and adjusting of different levels of granularity of provenance information during runtime. A provenance collection mechanism, which is

capable of accessing at the OS-level (for fine-grained provenance information), while at the same time it needs to have access to the workflow definition and execution information to capture high-level provenance information. Generally the desirable level of access for provenance mechanisms enables having an independent collection mechanism loosely-coupled to SWfMS (similar to O.S and Activity-level of access to workflow information). It collects retrospective and prospective provenance, because it has access to the entire workflow system such as workflow definition and execution information (similar to workflow-level). It requires having a combination of characteristics of each level of access to information in scientific workflow systems.

2. Architecture layers of provenance systems

As explained in section 5.1.8, provenance systems and provenance collection mechanisms could be accommodated in platform, framework and application layers in software architectural layers.

The collection mechanism in the platform layer (including file or database) captures very complete information when only one facet of behaviour is required. It is very difficult to reverse engineer all relevant behaviour of an application through its impact on file or database system such as a process of reconstructing causal relationships in collected provenance information. The collection mechanism at the platform layer collects provenance without intervention of the execution process and any major reconfiguration requirements for workflow systems. The collection mechanisms in the framework layer are sitting on top of the workflow systems and does not modify the source code of the application because the framework configuration specifies what provenance information is needed. It collects more useful information than platform layers but not as much useful provenance information as would be collected by the application layer.

The Application layer provides a complete view of one facet of behaviour without knowing the rest of application behaviours. It captures a narrower facet of application behaviour (in comparison with platform and framework). The Application layer is fully customized and collects everything that is required. The Application layer must have significant cooperation from the application to be tailored to what the application needs.

As discussed in section 5.1.8, workflow systems could be placed somewhere between application and platform layer. A framework is provided in workflow systems capable of taking pre-existing code and plugs into this code with some effort with minimum interference to code. It requires some configuration by the user. It is not as flexible as the application layer, which exploits source code.

3. Type of instrumentation

As mentioned in section 5.1.12, instrumentation determines the trade-off between having quality of provenance information and burdening the user, developer or even the performance of application. Post-runtime instrumentation collects provenance information when it is needed and mostly after the workflow execution from log files or databases. Post-runtime instrumentation uses user-annotation and scavenging instrumentation to reconstruct provenance information. In contrast, we are looking for runtime provenance collection instrumentation. This type of collection mechanism is integrated directly into the source code of provenance collection mechanism and might possibly impose a significant burden of workflow designer, programmer and application that run this mechanism at runtime; however, it captures higher quality and more precise provenance information with no configuration burden on users, which is desirable for scientific workflow systems.

There are four instrumentation methods introduced in section 5.1.12 (Compiler-based, Event-Driven Programming, Wrapper and non-functional concern). None of the methods have been used to capture adaptable workflow system collecting multiple-granularity provenance. The Compiler-based instrumentation method collects provenance mostly in operating system and some of provenance at the application level, defined in the context of SPADE [171]. Compiler-based instrumentation can capture fine-grained provenance at the level of system calls (operating System approach) and function calls (application approach). The provenance information at this level needs reverse engineering to reconstruction the causal relationship of collected provenance information. Importantly for this thesis, this approach cannot adjust the level of provenance granularity.

There are a number of SWfMSs that are instrumented by Event-Driven Programming for their provenance collection mechanisms, such as Kepler [46, 152], Trident [16, 202] and Karma [186, 187]. They are capable of capturing fine-grained (and coarse-grained) provenance information. The wrapper is another instrumentation method that captures coarse-grained provenance, introduced in ProvManger [58, 178, 206]. The wrapper mechanism is not adaptable in terms of capturing multiple-granularity provenance. The wrapper mechanism offers some sort of structure reconfiguration (such as adding or removing components) but not in the form of multiple-granularity adaptation.

The non-functional concern mechanisms (expressed in MOP or AOP) have not been used in the context of collecting and adjusting multiple levels of granularity of provenance. They enable the use of an adaptive provenance collection mechanism that is loosely-coupled to the SWfMS. They take a perspective from outside of the workflow system, thus they have access to the entire workflow system and its information. Therefore, they can collect both retrospective and prospective provenance. The non-functional concern mechanisms seem more fitted to adaptable provenance collection mechanisms and as a result in adjusting provenance granularity during runtime due to having access to the workflow-level (and possible activity-level) of information.

In this section, desirable design dimensions for adaptive multiple-granularity provenance collection mechanism have been explored. The key points of understanding from this section are the adaptability and multiple-granularity provenance features that we discussed in terms of instrumentation and level-of-access design dimensions. The desirable attribute of adaptability appears to be achievable through using non-functional concern instrumentation. The multiple-granularity feature requires a particular level of access to all levels of workflow and system information that is not supported in any of the introduced levels of access including workflow-, activity- and OS-levels.

We propose an adaptive workflow architecture in order to capture multiple-granularity provenance information. MOP and AOP are two software techniques that we use to implement our adaptive provenance collection mechanism presented in chapter 7. In the next section, an adaptive view over workflow architecture is explored for adaptive provenance

purposes. We focus on design viability of two adaptive provenance designs in workflow architecture that are inspired by features offered in the workflow controller.

6.2 Principles of adaptive workflow architectures

We introduce two adaptive designs for a workflow architecture that builds above the concept of workflow controller (explained in section 2.4). We investigate the possible places in workflow architecture that a provenance system can be accommodated to have adaptability and multiple-granularity provenance features. Following that, we show the design viability of clear and distinct adaptive provenance architectures for the provenance collection mechanism. In the next section, the provenance component is explored in the context of adaptive scientific workflow architectures.

6.2.1 Provenance component in adaptive workflow architecture

The provenance recorder contains a provenance collection mechanism able to adjust the level of provenance granularity (or able to support multiple-granularity provenance). In this section, we are looking for an appropriate place in workflow architectures to accommodate a provenance recorder component. In the workflow architecture presented in Figure 2.1, workflow instances are placed in SWfMS that contain a workflow engine and workflow controller (refer to section 2.3). The Workflow Engine runs workflow instances. The workflow controller communicates with the Information Service and workflow instances. It manages and models data movement and interaction with the environment of the workflow system. Therefore, we investigate which of the workflow instance, SWfMS, workflow engine or controller is the better place to accommodate the provenance recorder component.

The Kepler provenance architecture (refer to section 3.2.2 and Figure 3.3) accommodates the provenance recorder in the workflow instance. In this architecture, the provenance recorder can communicate with the workflow engine (which is a director). Therefore, it just captures a particular level of provenance granularity. Moreover, the mechanism of provenance collection in the provenance recorder does not directly collaborate with the workflow controller, because it is an independent entity that cannot interact with a workflow controller. Therefore, it cannot

benefit completely from the features offered by the workflow controller (explored in the literature review and in the next few paragraphs).

There is another approach presented in COMAD director. COMAD is a Kepler director that is considered as a workflow engine. The workflow engine (COMAD director) is accommodated in the Kepler SWfMS. In this approach, the collection mechanism is embedded in COMAD. COMAD, as presented in section 4.3.2, is capable of collecting fine-grained provenance because it is superimposed onto the rules of workflow semantics establishing interaction between workflow components. However, it is not able to adjust the provenance granularity. It also cannot communicate with environment that the workflow system is embedded in. COMAD in not capable of being adaptable and collecting different provenance granularity (being multiple-granularity provenance system).

Recall in section 2.3.3, the workflow controller is responsible (implicitly or explicitly) for looking after how the processes and tasks of workflows are executed and coordinated. It manages and models data movement through workflow processes; initialize, run and finish processes. Moreover, it can collaborate with entities in the computing environment (such as the Information Service) to get environmental information and connect into resources in the environment. Therefore, the workflow controller is not about the workflow itself, but also the environment in which is embedded. Therefore, the workflow controller is an appropriate place to employ adaptation mechanism into workflow instances exploiting information that is collected from the Information Service (refer to section 2.3). It also can collaborate with the workflow engine to manage extra functionality and non-functional behaviour for workflow system such as, adaptation, reconfiguration, monitoring, and provenance.

In the next section, we present two adaptive architectures designed on the basis of MOP and AOP adaptive software techniques. We provide clear designs for provenance systems in workflow architectures

6.2.2 Workflow Architecture in terms of provenance

We work towards a new design for workflow architecture that is motivated by our views of the applicability of separation on concerns in design of the workflow controller, with the aim of enable adaptability in provenance collection. In this chapter, we focus on the application of adaptive ideas in our provenance collection mechanism and show the design viability of our new provenance design in workflow architectures.

We design two provenance architectures that collect multiple-granularity levels of provenance information. The desirable properties of these architectures include adaptability, applicability and usability. The separation of our adaptive design techniques from the workflow architecture is a significant feature of our design. Our provenance collection architecture is an abbreviated form of a full production type system, which demonstrates the principles and viability of our approach.

As mentioned previously, we propose two distinct provenance architectures applying provenance collection mechanisms outside of SWfMSs', as presented in Figure 6.2 and 6.3. The provenance collection mechanisms in the workflow architectures are designed by adaptive software techniques (MOP and AOP). The differences and similarities and effectiveness of the MOP and AOP approaches in the implementation of adaptive provenance architectures are examined in chapter 7. Here we outline the broad designs of an adaptive provenance architecture using MOP and AOP.

Using each adaptive software technique, we are able to collect provenance information and propose some novel advantages, such as minimal modification of the application, low cost integration of the provenance mechanism into existing component-based workflow systems, and the capacity for adaptation - in this case adaptation of the granularity of provenance collected.

In the MOP and AOP architectures for provenance collection mechanisms shown in Figure 6.2 and 6.3, the underlying scientific workflow system is implemented in as a Process Network Application (PNA), as discussed in section 3.4.

In Figure 6.2 and 6.3, the Information Service (explained in section 2.3) provides required information for the workflow controller to collaborate with external entities such as the computing environment and users. The Information Service provides variety of information ranging from user requests and requirements, to the condition of underlying infrastructure resources and network information. The workflow controller would be informed about the level of provenance required based on received information from Information Service. The workflow controller informs the provenance recorder to adjust the level of provenance granularity to collect requested level of provenance. The provenance information that should be collected in each level of provenance granularity is represented in the corresponding MoP.

In this section, two adaptive architectures facilitated by MOP and AOP software techniques are presented as provenance collection mechanisms. The constituent components of the architectures are explained. In the next section, the design of a MOP for reflective provenance collection mechanism is demonstrated. The AOP design is discussed in section 6.4.

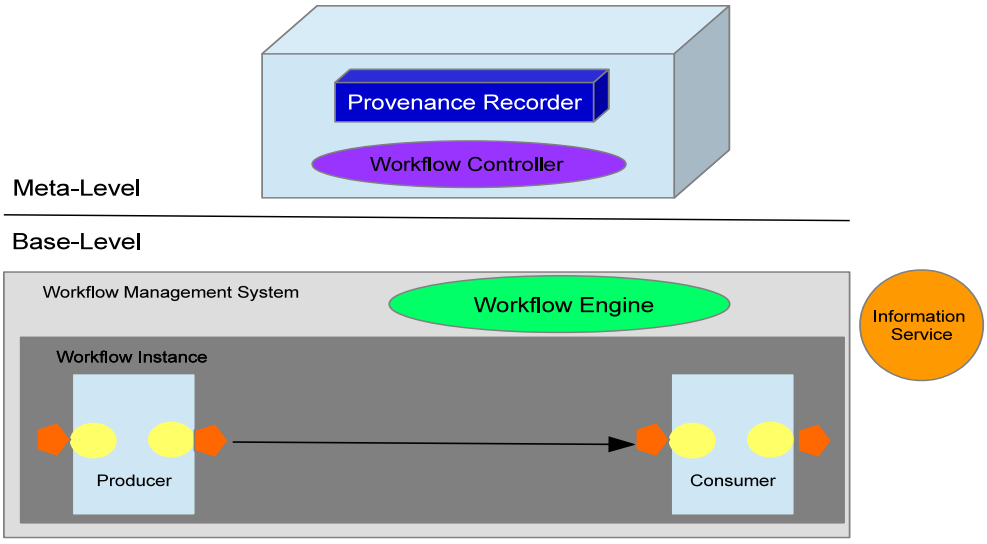


Figure 6.2. MOP architecture for provenance collection mechanism.

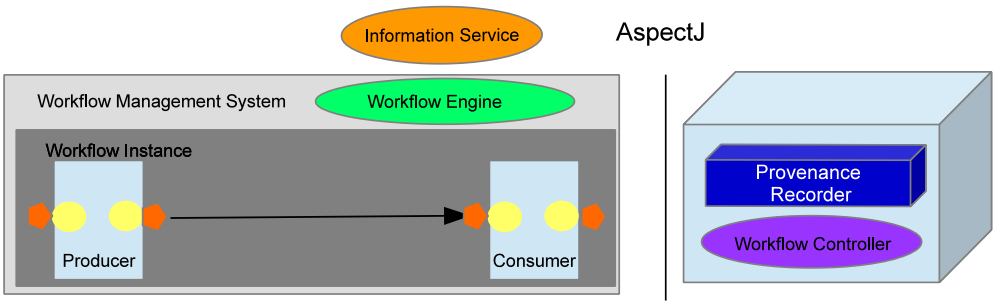


Figure 6.3. AOP architecture for provenance collection mechanism.

6.3 A MetaObject Protocol design for adaptive provenance in Scientific Workflow

In this section, we propose a design for an adaptive provenance architecture using a MOP and a reflective architecture aiming to provide provenance-collection, distribution and adaptive-granularity meta-behaviours.

As described in section 2.4, the MOP relies on two fundamental concepts of reification and reflection for a meta-behaviour in order to define collaboration and communication between base-level and meta-level objects. Reflective systems separate the functional and non-functional behaviours, which define, respectively, what the program does and details of how it does it. This separation of the functional and non-functional behaviours in reflection, which is defined using a MOP technique, allows programmers to focus on the specific behaviour of system and customize through the MOP. This non-functional behaviour, known as meta-behaviour, is introduced through by way of a MOP. MOP in a programming language context is a mechanism for abstraction that divides programs into base and meta-level programs, enabling users to control critical aspects of implementation strategy without being exposed to implementation details [117].

MOP utilises two interfaces, a base-level interface and an interception (adjustment) interface for base- and meta-level, respectively. A meta-level program with interception interfaces (as explored in section 2.4.1.2) can control and customize the semantics (or behaviours) and implementation of underlying systems implemented in base-level program [117] and presented in base-level interfaces. There are four design principles that need to be taken into account in order to design meta-level interfaces [117]

- *Scope control*: control over the identified scope of base-level objects needs to be restricted for meta-level interfaces.
- *Conceptual separation*: meta-level interfaces should customize particular aspects of the implementation without a requirement for understanding the whole system.
- *Incrementality*: customization and implementation of some aspects of the system (implementation) should be separately handled without having to customize the rest of the implementation.
- *Robustness:* bugs in a meta level program should have limited effects on the rest of system.

In this work we consider a scientific workflow system as the base-level application represented in a form of a PN framework (PNA) described in section 3.3 and 3.4. As mentioned in section 3.4, the meta-level consists of meta-objects containing a number of either structural or behavioural concerns of the base-level objects known as meta-behaviours. It is possible to assign a number of meta-behaviours to a meta-object. The selected meta-behaviours directly influence on the design process of a MOP. In the following section, we explain provenance-collection, distribution and adaptive-granularity meta-behaviours for scientific workflow systems, respectively, in section 6.3.1, 6.3.2 and 6.3.3. In the next section, a MOP is defined for Provenance-collection meta-behaviour.

6.3.1 A MOP for Provenance-collection meta-behaviour

In this section, a MOP is proposed for adaptive provenance collection mechanism in the architecture presented in Figure 6.2. In this design we just consider the Provenance-collection meta-behaviour for our design, as shown in Figure 6.4. Each object at the base-level has an associated meta-object with different interfaces (meta-behaviours).

A provenance-collection meta-behaviour observes the behavioural and structural aspects (concerns) of the workflow constituent objects. The structural aspect of this meta-behaviour is related to the structure of workflow constituent objects (including processes, channels, threads and ports); the workflow graph structure, the workflow engine and the controller. The behavioural aspect of the provenance-collection meta-behaviour is related to communication between base-level objects in the underlying workflow system. Communication can be considered in different level of detail, such as the data passing between components to establish the communication, or even a communication style in the form of method invocation. Therefore, behavioural aspects of meta-behaviours are expressed and found in method invocation (message passing) between base-level (workflow constituent) objects during workflow execution. This behavioural aspect of provenance-collection meta-behaviour is captured in the retrospective phase (refer to section 5.1.4) of provenance collection. The structural aspect is captured in the prospective phase (refer to section 5.1.4) of provenance collection that captures the specification and configuration of workflow. The designed MOP in Figure 6.4 has just the provenance-collection meta-behaviour as a reflection mechanism to

observe the static and dynamic provenance behaviour of system. In the following sections, we discuss other meta-behaviours as shown in Figure 6.5.

As explained in section 2.4.1.2, a MOP determines reflection and reification mechanisms between base-level and meta-level for collaboration and communication purposes. In section 2.4.1.2, meta-behaviours are introduced to the system through MOP mechanisms (reification and reflection). We explained the provenance meta-behaviour that is declared in reflection mechanism of a MOP. Now, we discuss the reification mechanism that identifies how the structure and behaviour of base-level objects are transformed into first class manipulable and computable objects (meta-level objects), as discussed in section 2.4.

There are two structural and behavioural models of reification. In our design, scientific workflow systems are considered as the base-level application. In this regard, structural reification transforms (reifies) the structural aspect, includes creation and existence of workflow constituent objects (including processes, ports, channels and threads) into a form of meta-object. While, the behavioural reification is representative of a system's strategy and determines how to act on reified method invocations. The behavioural aspect includes the method invocations between workflow constituent objects that determine the relations of objects in terms of provenance dependencies during runtime.

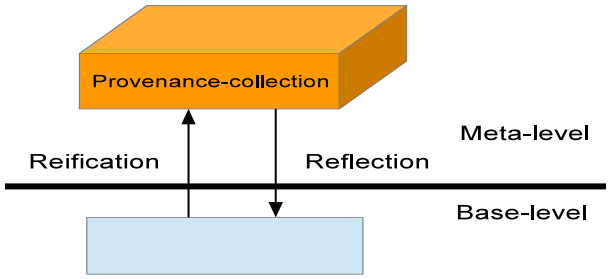


Figure 6.4. Provenance meta-behaviour.

In this section, a MOP is designed in a form of representing provenance-collection meta-behaviour. Our reflective architecture is oriented specifically to focus on the intersection between reflection and provenance. While workflow systems may require having other reflective meta-behaviours (figure 6.5) such as a distribution meta-behaviour that is an important aspects of scientific workflow, as explored in section 2.2.

As shown in figure 6.5, we are going to add other meta-behaviours (such as distribution and adaptive-granularity) to our reflective system that has only provenance-collection meta-behaviour. Therefore, a new MOP needs to be designed for each new meta-behaviour to expose it to users and programmers. The design of MOP depends on design principles of reflective system including structural and behavioural reification approaches; the considered meta-behaviours of systems in introspection and interception mechanisms; and decisions about reification time (compile or runtime).

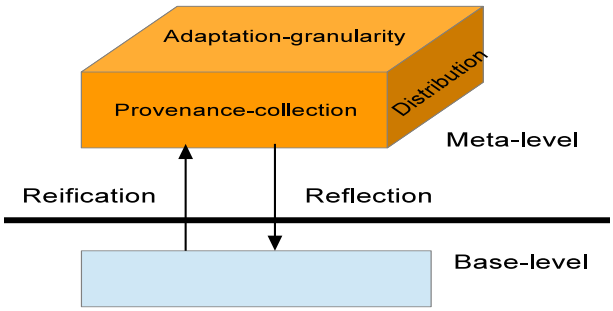


Figure 6.5. Meta-behaviours.

Some meta-behaviours such as monitoring, can easily be defined similarly to previous MOP design; however, some others including distribution meta-behaviour needs fairly different design from what we have done in this section for provenance-collection meta-behaviour. In the next section, we firstly explore the design principles of a meta-level application. We will use the design principles as guide to our design of meta-behaviours. In the following, we will present the design of a distribution meta-behaviour alongside with provenance-collection and adaptive-granularity meta-behaviours.

6.3.2 A MOP for Distribution meta-behaviour

We introduce our provenance architecture in scientific workflow systems that are running in different distributed computing environments as shown in Figure 6.1. The execution of workflow systems and activities on distributed environments has many complications, some of which are expressed in section 6.1. These include fluctuations in: availability of resources, resource performance and network limitations. Therefore, a mechanism (similar to reflective component-based middleware, explored in section 2.4.1.5) is required making the

heterogeneity of underlying distributed system transparent from the users and application's perspective. We define distribution as a meta-behaviour for this purpose

The distribution meta-behaviour aims to make the execution of workflow constituent objects on distributed environments transparent from the base-level application point of view. A Meta-object contains complete information about method invocations (because the method invocation are reified). One of the pieces of information is about where the target base-object of method invocation is located (the locality of target object). The distribution meta-behaviour enables implicit distribution of base-objects by modifying the locality of target object. This distribution meta-behaviour requires intercepting a method invocation and transmitting it to a possibly remote host where the target base-object of method invocation is located. The implementation of distribution meta-behaviour requires sophisticated mechanisms (the details of which are beyond the scope of this work) to apply it to workflow systems. Enigma [111] implements the distribution meta-behaviour. Enigma is facilitated by a generic mechanism of message decomposition that makes it a very flexible and makes sit possible to introduce and add new behaviours into systems. Therefore, we take the Enigma design to implement our desire meta-behaviours in chapter 7.

The distribution meta-behaviour is designed in the Enigma meta-level application. We presented design principles of our MOP for provenance meta-behaviour in section 6.3.1; however, we can also design and implement it with the mechanisms offered in Enigma. In the next section, Enigma and its mechanisms are presented to provide sufficient background for understanding of how we use it to design and implementation provenance and distribution meta-behaviours.

6.3.2.1 Enigma

As explained in section 2.4.1.6, Webb *et al.* [111] developed a MOP called Enigma to introduce Grid-related behaviours for an application. Enigma is implemented in Java, and can be used to reify objects in an object-oriented system. It was originally created to support the PAGIS [111, 132] middleware system. PAGIS is a process network framework that operates similarly to PNA, as presented in section 3.4. Enigma [111] is a meta-level application that structurally reifies objects and classes; and behaviourally reifies method invocations. In

addition, it enables customization of objects and reification of method invocations. Enigma introduces three phases of message-based communication (decomposition phase) in order to customize each message. These phases are called the marshal, transmit and execute phases, as shown in Figure 6.6

- The *marshal phase* prepares messages that should be sent a target meta-object.
- The *transmit phase* coordinates the message receiving process with the target meta-object.
- The *execution phase* applies the message on top of the target base-object [111].

Enigma's decomposition phase allows the composition of new meta-behaviours into each phase, so it allows interception and customization at each phase. The flow of a method invocation through the Enigma MOP in the context of a case study is explained in appendix G. A number of meta-behaviours can be applied to each phase of Enigma message decomposition, such as the compression of large method parameters and results during the marshalling phase; distribution meta-behaviour on transmission phase; tracing [111] , and debugging and provenance-collection meta-behaviour on execution phase.

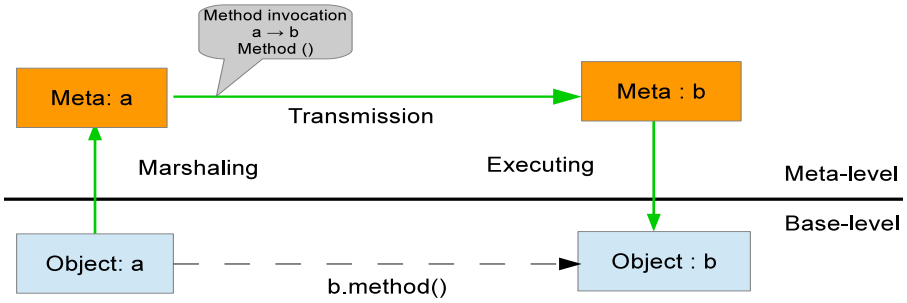


Figure 6.6. A model reifying the three phases of method invocation [111].

A workflow system (dataflow) is constructed with a number of components and is executed under different execution models. Process Network (PN) is one of these types of dataflow execution models defined in [145] and implemented in [147]. In a conventional model of reflection, each meta-object presents a reification of a corresponding object at the base-level, and the meta-level reifies communication between objects. Therefore, each constituent object of PN has a corresponding meta-object and communication is reified between base-objects at the meta-level.

A model proposed [111, 145] and implemented in Enigma [111] that aggregates the reification of process, thread of process, input port(s) and output port(s) into one meta-object

that is called Computation meta-object or meta-Computation. Thus, meta-Computation is the meta-level representative of several process network objects (base-objects) [111]. In this mechanism, a process network model is viewed as a network of autonomous computation resources that are connected by unidirectional links. The Computation meta-object is inserted in the transmission phase of relevant meta-objects in the Enigma MOP, as shown in Figure 6.7.

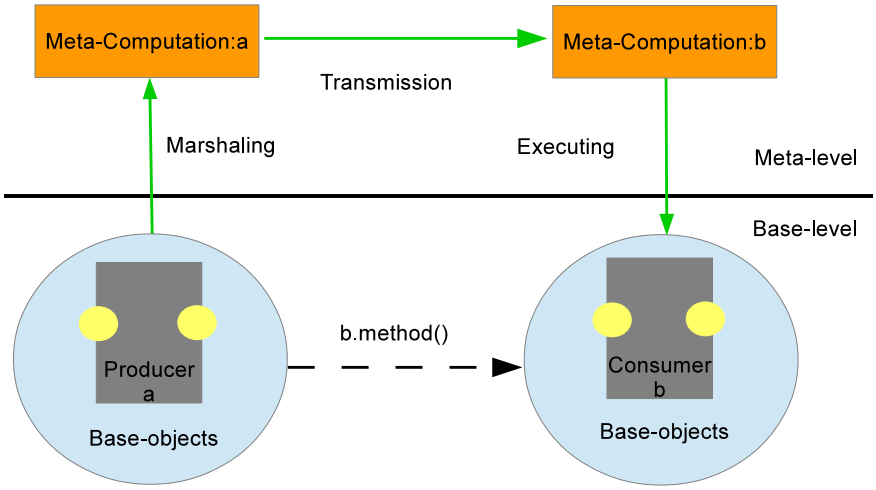


Figure 6.7. The Computation meta-object reifies a computation's components [111].

In this model, each Computation meta-object represents a number of base-level objects and each meta-level object represents the reification of Computation meta-object and its communication. This model efficiently customizes (and configures) all the aspects and elements of a Computation meta-object, instead of customizing each element of base-level objects individually. Moreover, it provides a convenient mechanism to customize and introduce new approaches to expose customization and reconfiguration operations on PN constituent objects. Reconfiguration and customization enables dynamic modification of a PN applied to the Computations meta-objects. As such reconfiguration and customisation perform on a meta-meta-level layer that considers the Computation meta-object as base-object, as shown in Figure 6.8 [111]. Therefore, the Computation meta-objects in the meta-level can be customized and reconfigured in another meta-layer (meta-meta-layer) that treats Computation meta-objects as base-objects. This approach offers a simplified view but, due to the abstraction of PN objects into computation objects, it is not capable of fine-grained customization of individual base-level objects [111].

We provide a case study of a simple example in Enigma in appendix G. In this case study, the mechanism for method invocation in the Enigma MOP is explained. The implementation of Enigma is presented in chapter 7.

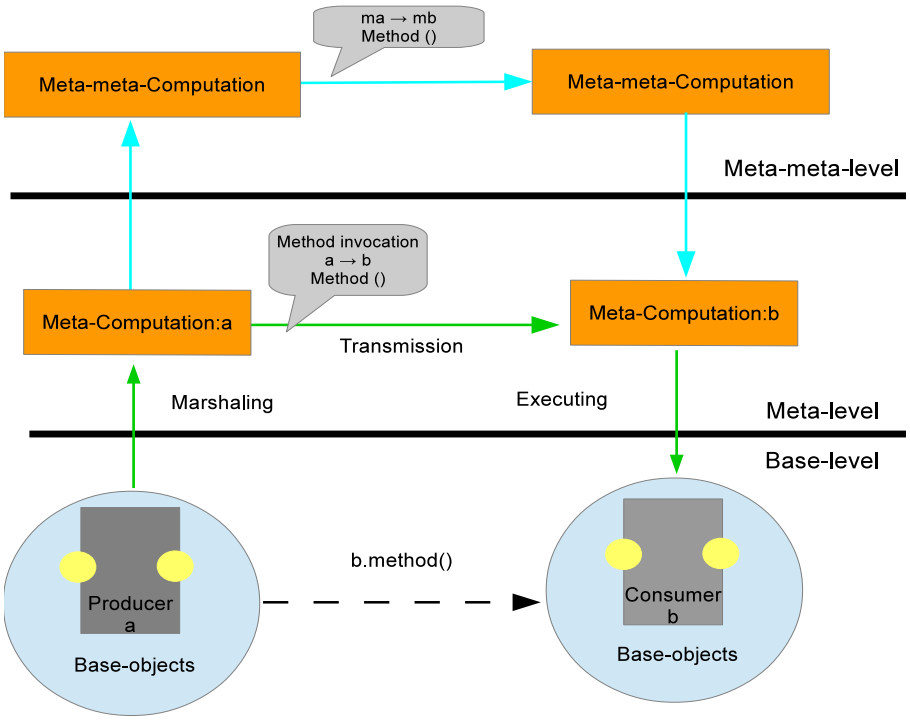


Figure 6.8. Meta-meta-level for customizing all computation components[111].

6.3.2.2 A MOP for Provenance-collection and Distribution meta-behaviours in Enigma

As discussed, Enigma's message decomposition phases allow composition of new meta-behaviours into each phase. In this regard, several meta-behaviours can be applied to each phase of Enigma's message decomposition at the same time. Enigma's message decomposition offers a suitable architecture in which to apply meta-behaviours, such as compressing meta-behaviour, distribution meta-behaviour, tracing [111], and debugging. The distribution meta-behaviour is applied in the transmit phase.

It is possible to apply the provenance-collection meta-behaviour to the execution phase, as a mechanism to inspect the workflow constituent components and collect required information. We describe the implementation of provenance and distribution meta-behaviours on Enigma

in chapter 7. Adjusting the provenance granularity is one of the main focuses of this thesis. We define it in a form of a meta-behaviour in the next section.

6.3.3 A MOP for Adaptive-granularity meta-behaviour

There are two ways to view and apply a MOP to adaptive-granularity collection of provenance information - either through existing provenance meta-behaviour or to define it as a separate meta-behaviour.

In the first approach, the provenance-collection meta-behaviour has an introspection interface to observe the workflow behavioural and structural behaviours of the workflow's constituent objects. This interface would observe the underlying workflow system and collect provenance information. It is possible to define an interception interface on this meta-behaviour to adaptively collect multiple-granularity provenance. These interception mechanisms (applied to workflow controller to) change the behaviour of the provenance system and force the provenance collection mechanism to collect at the defined level of granularity in the workflow controller. The controller is notified by the Information Service to change the level of provenance granularity (acting as a trigger for adaptation).

The concept of adaptation in this approach is a computation about which computation to pursue next that is performed from another entity (i.e. the workflow controller). This meta-computation makes some changes in the policy of provenance collection mechanism, for example changing from a coarse-grained collection mechanism to a fine-grained collection mechanism. This meta-Computation reasons about controlling mechanisms to preserve the status and behaviour of data items and activates processes when specific events occur. We implement this integrated adaptation approach on provenance collection mechanisms in chapter 7.

The second approach is defining a separate adaptive-granularity meta-behaviour. This meta-behaviour adjusts the provenance granularity of the collection mechanism according to the information provided by the Information Service. It is applied to the provenance-collection meta-behaviour and enforces the requested granularity level on it. This meta-behaviour

intercepts (modifies) the behavioural aspect of provenance-collection meta-behaviour. Therefore, the adaptive-granularity meta-behaviour should be accommodated in a meta-level above the provenance-collection meta-behaviour (meta-meta-level).

The integrated adaptive-granularity meta-behaviour in the first approach is applied to the defined MOP for the provenance-collection meta-behaviour by defining an interception mechanism. While the second approach defines a separate meta-behaviour. Diagrammatically, the second approach provides another meta-layer on top of the existing meta-layer (as shown in Figure 6.8) to apply adaptation. The second approach, using a meta-meta-layer clearly adds the overhead of an additional MOP implementation, so we avoid this approach when possible. Therefore, we implement the first approach in the next chapter.

In this section, we described provenance, distribution and adaptive-granularity meta-behaviours for the adaptive provenance architecture presented in Figure 6.2. We introduced our meta-behaviours' design in the context of Enigma. In the next section we explore an adaptive provenance architecture using the AOP technique. We define and design provenance and adaptation-granularity aspects in the adaptive provenance architecture based on AOP the software technique.

6.4 AOP design for adaptive provenance architecture in SWF

In this section, we are aiming to design an adaptive architecture for provenance collection mechanism based on the AOP software technique. In this architecture shown in Figure 6.3, we define two aspects: provenance-collection and adaptive-granularity for this architecture. . These are performed on scientific workflow system implemented in PNA.

As mentioned in section 2.4, AOP is an adaptive software technique. AOP provides separation of cross-cutting concerns to increase modularity [69, 70], as explored in section 2.4.2. AOP enables designer or developers to structure programs in way that represent the way they think about the programs [70]. AOP is integrated to existing technologies to provide additional mechanism to address cross-cutting concerns in design and implementation. It also collaborates with existing programming paradigms (and languages) instead of replacing them, for example AOP has been integrated very well into OOP and component-based systems, as

explored in section 2.4.2.1. An AOP application includes functional concerns and aspects. The functional concerns are the implementation of the application's business logic in the form of classes or components. Aspects are the programming unit responsible for implementing the application's cross-cutting functionalities such as, transaction, security and logging that exist in multiple places and entities in a program [69].

Aspects in AOP are not the same as meta-behaviours in MOP but they achieve a similar purpose. Aspects and meta-behaviours express and separate areas of interest in a program. Therefore, the ability of expression and separation of aspects and meta-behaviours in AOP and MOP provides separation of concerns. Aspects and meta-behaviours represent concerns of a program. Meta-behaviours provide a view of concerns in a meta-level, as shown in Figure 6.2, while aspects provides a view of concerns that are addressed across the system in an aspect programming unit, as shown in Figure 6.3.

As explained earlier, an AOP application includes the functional application and aspects. In our design, we consider the PNA (workflow system) as the functional application (the implementation of the application's business logic). As explained in section 3.4, the functional application is a PN framework designed along component-based design principles. We design provenance-collection and adaptive-granularity aspects proposed for an AOP oriented adaptive provenance architecture. The designs of the desirable aspects are performed on scientific workflow systems implemented in PNA. In the next section provenance aspects are explored. In the following we will use the terminology of Aspects and AspectJ that we explained in section 2.4.2.2.

6.4.1 Provenance-collection Aspects

A provenance aspect aims to collect provenance information based on a specified Model of Provenance (MoP). The places (points) in the workflow system (program) where provenance should be collected by provenance aspects are related to the MoP. The provenance aspect defines a number of pointcuts (explained in section 2.4.2.2) on the components in PNA application. The pointcuts determine points in the program where an aspect is aiming to collect provenance information. Therefore, pointcuts explicitly determine the specific

provenance information that we are aiming to collect. In this model, we define and use a number of pointcuts to implement the MoP. From section 4.5.2, we recall that three models of fine-, medium- and coarse-grained MoP are defined that identify the desired level of provenance granularity. Thus, for example, we require more pointcuts for fine-grained MoP in comparison with coarse-grained MoP because fine-grained MoP requires the collection of more information from more points in the program.

The design of pointcuts is directly related to the structure of components (processes) in the PNA application presented in section 3.4. For example a pointcut can be defined on the "fire()" method as presented in Figure 6.9. It presents an aspect upon the fire method of processes in PNA. PNA classes are shown in a blue box, and an aspect is shown in a red box. The "fire()" methods are joinpoints (explained in section 2.4.2.2) that construct a pointcut named "ProcessFire()". This pointcut has a "before" advice (explained in section 2.4.2.2) that performs the logic of the advice before the fire methods (joinpoints) in all processes regardless of their return type and parameters.

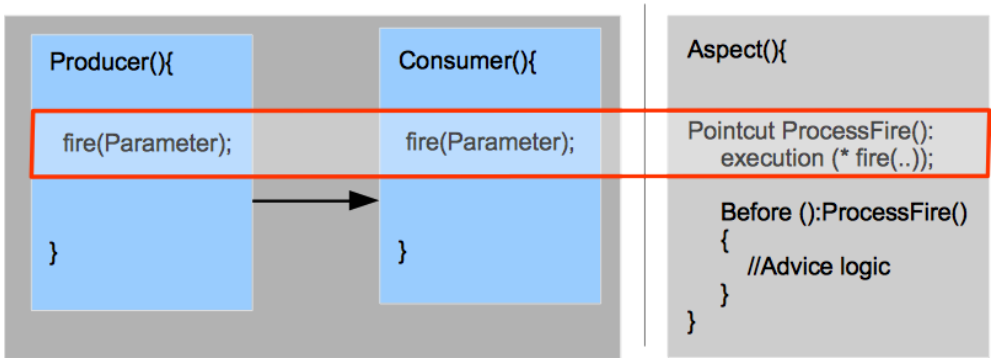


Figure 6.9. Aspect-oriented programing case study in a process network.

The MoPs that guide the application of pointcuts are designed in a format compatible with the OPM framework for MoP. For example the medium-grained MoP (defined in section 4.5) needs at least five pointcuts for each OPM dependency - WasGeneratedBy(), Used(), WasTriggeredBy(), WasControlledBy(), WasDerivedFrom(). Consequently, the provenance aspect inserts a pointcut for each OPM dependency, and the required provenance information for these dependencies is collected in an advice of these pointcuts. For example, the "Used()" OPM dependency requires information regarding data that is used in the process. A pointcut is defined for this dependency on the "fire()" method of processes on PNA applications as presented in Figure 6.9. For this pointcut an advice is defined that collects the required

information (such as artifact and process information) before the execution of the “fire()” method. For other MoPs with finer or coarser grained levels of provenance collection, a different level of information is collected. We will explain our implementation of adaptive-granularity aspect in section 7.3.1.

6.4.2 Adaptive-granularity Aspect

We define all the required pointcuts for each MoP in one aspect. Multiple-granularity MoPs in this thesis are concerned with different levels of provenance granularity as presented in section 4.5.2 such that three MoPs are defined (fine-, medium- and coarse-grained MoP) in an aspect. The adaptive-granularity, in this design, is handled through an advice method of pointcuts. Three methods are defined in each advice method corresponding to each level of provenance granularity identified by the Information Service. For example, when the “Used” pointcut is activated and it runs its advice, there are three methods for constructing the “Used” dependency (places in advice logic in Figure 6.9) that are selected to the requested level of granularity.

The Information Service collects environmental information and user requests and then decides which MoP (level of granularity) should be considered for provenance information. In the case of changing the requested level of provenance, another MoP should be selected and activated. Therefore, a proper method in the advice is selected in accordance with the MoP and level of provenance.

All the pointcuts in an aspect are always active. The pointcut is always performing on a particular joinpoint during program execution. As the program execution reaches the point that a pointcut is identified, the advice method of that pointcut is activated. For example, there if a pointcut is referring to a method such as the “fire()” method. The aspect weaving is set and configures a pointcut to the application object (either during runtime or compile-time). Therefore, we cannot de-active or disconnect them during program execution. Consequently, the adaptation mechanism has to be applied in an aspect‘s advice, which can be selected at runtime instead of pointcuts, which can’t, in themselves, be adjusted.

In this design for adaptation, the Information Service determines level of provenance granularity. The adaptation mechanism decides about running the collection methods in an advice based on the information received from the Information Service. In this design, all the pointcuts are active but only the advices on requested aspects are able to collect provenance information to be stored in provenance storage. We will explore our implementation of adaptive-granularity aspect in section 7.3.2.

6.5 Summary

Many workflow systems required to run on distributed environments are faced with changing computing environments. Therefore, it is necessary to have adaptive provenance mechanisms in scientific workflow systems to deal with incoming changes in the environments. A number of possible scenarios presenting several changes in computing environment were presented in this chapter; however, in this thesis we focus on adaptive provenance collection mechanisms to adjust the amount of captured provenance. In this approach the granularity of provenance information is changed to adjust the amount of captured provenance based on provided information from the Information Service, which informs of the environmental changes in the underlying infrastructure and network situation of a workflow system environment.

We use separation of concerns as the guideline behind the principle concepts of our adaptive provenance architecture for scientific workflow systems. The separation of concerns is expressed in both reflection (MOP) and AOP software techniques. We have explored the design principles of reflection to design MOP for adaptive provenance architecture with provenance-collection, adaptive-granularity and distribution meta-behaviours. The meta-behaviours were designed based on design principles of Enigma. AOP is the other software technique that we employed on design adaptive provenance architecture. The implementation of both AOP and MOP oriented adaptive provenance collection architecture are presented in chapter 7.

7 CASE-STUDY: ADAPTIVE PROVENANCE COLLECTION IN A WORKFLOW SYSTEM

In chapter 6, we presented two designs for adaptive provenance architecture aiming to adaptively collect multiple-granularity provenance in distributed scientific workflow systems. We used MOP and AOP software techniques to design adaptive provenance collection mechanisms. In this chapter we are going to show an implementation of these adaptive provenance collection mechanisms. Our PNA framework (explained in section 3.4) is used as the underlying scientific workflow system in the implementation.

In section 6.3, we showed the use of MOP techniques in designing three meta-behaviours – provenance-collection, distribution and adaptive-granularity. We are going to implement these meta-behaviours is the Enigma MOP. In this implementation, described in section 7.2, PNA is a base-level application with Enigma on top of it as a meta-level application. This allows us to separate the functions and behavioural concerns.

In section 6.4, two aspects - provenance-collection and adaptive-granularity - were designed using AOP software techniques. We are going to implement these aspects in an aspect program implemented with the AspectJ framework. In section 7.3, this aspect program is integrated with PNA acting as a functional application in this implementation. Finally in this chapter, we examine and evaluate the implemented provenance collection mechanism, some of the design dimensions that are presented in chapter 5.

7.1 Experimental Configuration

In this section, we provide a scenario explaining how an adaptive provenance collection mechanism works over a computing environment. The collection mechanism adaptively adjusts the level of provenance granularity to satisfy the user requests in terms of budget, deadline and provenance policy. Work is underway to move this implementation into a distributed computing environment (we are aiming for integration in the context of Google Cloud) and complete experimental evaluation.

Consider a situation in which a scientific workflow system is run over a Grid environment that is facilitated by a provenance system. We assume the market-based (or economy-based) Grid [34] should consider the economy factors as a QoS user requirement. In our market-based mechanisms, time, cost, and provenance collection policy, which a user needs, are the important QoS factors. Deadline and budget are the most common economy factors as QoS user requirements. The environment has resources (with various prices) and provenance collection service that their prices vary based on demand for the resources.

Our case study is an illustrative scenario in which to explore concepts of adaptation; the estimates used will of course be different in practice. Suppose that for the purpose of our case study, we use prices in the range of 10 to 20 Grid$ per hour (Grid$ is Grid currency, following the notion introduced by Buyya *et al* in [34] and used in our work [205]). The cost of provenance collection (Grid $ per hour) is a constraint in the computing environment that will trigger adaptation. Suppose further that the cost of collecting fine-grained provenance (8 Grid $ / hour) is greater than medium-grained (4 Grid $ / hour) and coarse-grained (2 Grid $ / hour), because it collects a greater amount of provenance information, expending more computing and storage resources.

In this scenario, users submit a scientific workflow experiment with some configuration information (as was used in [205]) including budget (we use budget in the range of 2000 to 10000 Grid $ [205]), deadline (we use deadline in the range of 200 to 600 hours [205]) and provenance policy to the Information Service, which is an entity in the workflow system architecture explained in section 2.3. The Information Service enables users to submit their requests for customizations through the SWfMS while it has environmental information including the condition and price of underlying infrastructure resources and network information in terms of channel bandwidth (for the purpose of this case study we use bandwidth in the range of 100 to 2048 Mbps).

The Information Service informs the provenance system during workflow execution. It updates its environmental information and user requests as they change. We envisage that this will expose useful provenance policy models that enable a user to distinguish the extent of provenance collection. The user can choose one of three following provenance policies through Information Service,

- *No Provenance:* the user does not want provenance data.
- *Specific Level of Provenance*: the user wants a specific level of provenance granularity, including fine-, medium- and coarse-grained granularity.
- *Adaptable Provenance*: the user wants the best possible level of provenance granularity that fits in the assigned budget.

The Information Service informs the scientific workflow system about provenance policy. We investigate the possibility that users specify adaptable provenance, and we make use of proposed adaptive framework to optimize choice of granularity according to the underlying environment. If the user selects a specific level of provenance, the requested level of provenance granularity is collected based on assigned budget. The budget determines the maximum user payment for completion of the workflow experiment and receives the requested level of provenance information in accordance with experiment's results. Changes in the price of resources during the workflow execution might cause the cost of execution and provenance collection to exceed the assigned budget. In this case, the execution of the workflow experiment is cancelled.

If the user selects adaptive provenance from the provenance policies, the adaptive provenance collection mechanism selects the level of provenance granularity according to the price of resources, cost of collection and the assigned budget. The adaptive provenance collection mechanism uses multiple-granularity MoP in this provenance policy, thus it is capable of adjusting the level of granularity. For example, if the cost of provenance is less that the budget, the adaptive provenance collection mechanism collects fine-grained provenance. However, if the changes in resources' price make the cost greater than the budget, the adaptive collection mechanism adjusts the level of provenance granularity according to these changes (for instance, it collects medium-granularity provenance). Environmental information and user requests are updated frequently in the Information Service. If they are changed, the Information Service informs the adaptive provenance collection mechanism to adjust the level of provenance granularity. The Information Service makes the changes in the execution environment transparent from the point of view of the provenance system.

We provide a proof of concept study as a first step to realization of the scenario above, focusing on the mechanism of adaptation. The scenario demonstrates a simplified case in which an adaptive workflow system incorporating an adaptive view of provenance that can

respond to changes in environment. We aim to study ways of constructing such a system to provide a demonstration of the design and implementability of the concept.

In the following, our case study of the scientific workflow system is implemented on top of the PNA framework presented in section 3.4.2 (Figure 3.10). We implement MOP and AOP oriented adaptive provenance collection mechanisms in our case study, as shown in Figure 7.1 and 7.12 respectively. Both implementations of the adaptive provenance collection mechanism in this thesis are informed by the Information Service and are capable of collecting provenance at multiple levels of granularity, as shown in Figure 6.2 and 6.3.

7.2 A MOP oriented adaptive provenance collection mechanism

In this section, we present a MOP oriented adaptive provenance collection mechanism aiming to collect (either in multiple-granularity or specific level of granularity) provenance in scientific workflow systems as explained in chapter 6. This MOP oriented mechanism has provenance-collection, distribution and adaptive-granularity meta-behaviours, which are explored in this section.

In this thesis, we have used a process network graph to explain several concepts including a MoP (shown in in Figure 4.1) and a MoC (shown in Figure 3.4). The implementation of this process network in a PNA framework is presented in section 3.4.2 (Figure 3.10). The process network is used as a case study representing the underlying workflow system and is shown in Figure 7.1.

The case study for this chapter has a main class (named “PNATest”), which constructs the PN structure and initializes the required level of granularity at the beginning of program, as shown in Figure 3.10. There is “trigger()” method in this main class that starts the execution of the workflow graph, after creating PN components (base-level objects), establishing the network and the meta-objects. The process of execution and method invocations in Enigma are explained in a simple case study and presented in appendix G. In this section, we show how Enigma is configured and customized to operate on top of this process network. Our case study implemented using MOP is shown in Figure 7.1.

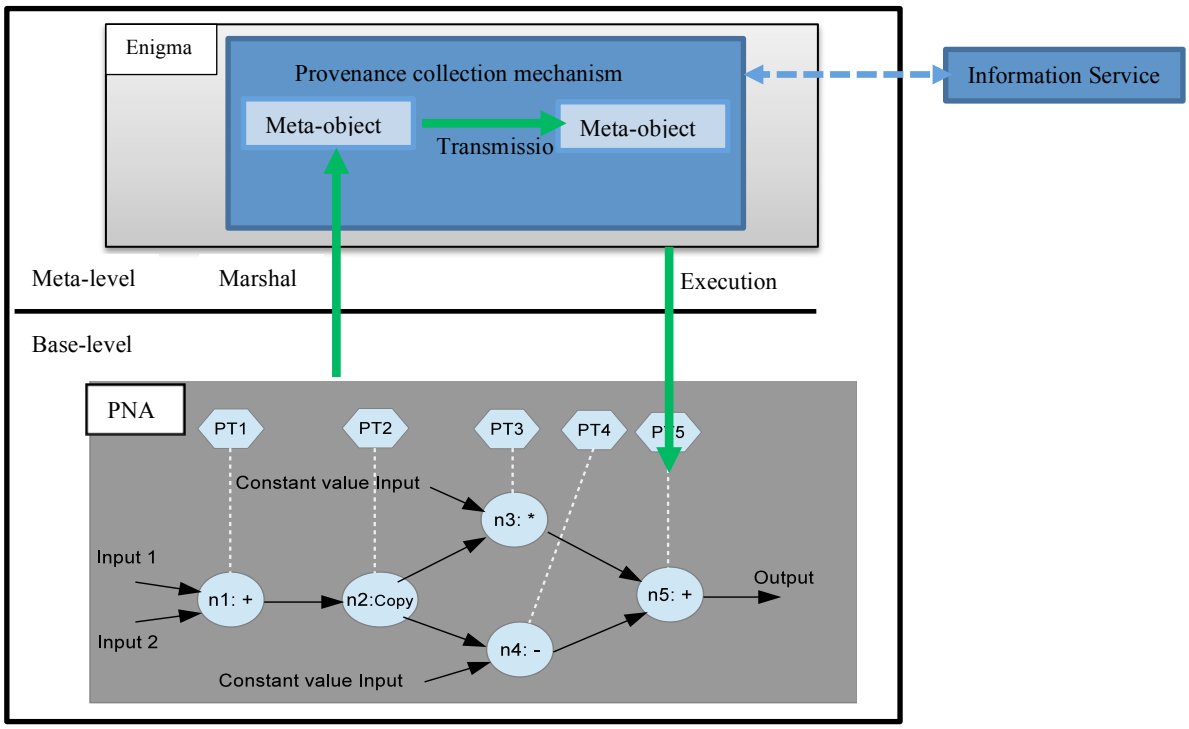


Figure 7.1. Case study architecture: MOP viewpoint.

7.2.1 Enigma MOP

Our adaptive provenance collection mechanism and its meta-behaviours are designed based on the Enigma meta-level application as explained in chapter 6. The design principles of Enigma were presented in section 6.3.2.1. In this section, the implementation of Enigma is explored to provide sufficient background to understand how it implements the meta-behaviours.

As mentioned in section 6.3.2.1, Enigma is a MOP that enables messages decompositions (method invocation) and composition of meta-behaviours. It decomposes message-based communication (method invocation) into marshal, transmit and execute phases as shown in Figure 7.1. These phases represent the structural reification of base-level objects. The reification of these messages decomposition phases enables representation of meta-behaviours in a form of first-class objects (explained in section 2.4.1.2) for meta-level programmers. Enigma allows composition of new meta-behaviours through customization of one or a number of phases and represents meta-behaviours as first class objects.

Figure 7.2 shows an Enigma UML Class diagram containing core interfaces and a number of classes related to our implementation. In Enigma, structural reification, which includes creation and existence of workflow constituent objects (as explained in section 6.3.2.1) into a form of meta-objects, is handled by “MetaObject” and “MetaMessage”, as shown in the Enigma UML class diagram in Figure 7.2. “MetaObject” is the structural reification of base-level objects (base-objects) and “MetaMessage” is the structural reification of messages transferred between base-level objects.

Meta-level programmers are able to add meta-behaviour to each message decomposition phase. “MetaHandler” is a representation for a meta-behaviour that can customize the message decomposition phases. As shown in Figure 7.2, “MetaMarshalHandler”, “MetaTransmitHandler” and “MetaExecutionHandler” interfaces are three specializations of the “MetaHandler” interface. The interfaces enable meta-level programmers to implement meta-behaviours, such as provenance meta-behaviour that is implemented on “MetaExecutionHandler”.

The “MetaMarshalHandler” interface has a method named “handlerMarshal” that is implemented by meta-level programmer in order to implement the marshal meta-behaviour (explored in section 6.3.2.1). This meta-behaviour is added to the “MetaObject” that sends the message (by “addMetaMarshalHandler” method). As a result, meta-level programmers are able to manipulate all the messages that are sent by source base-level objects to target base-level object.

The “MetaTransmitHandler” interface has a method named “handlerTransmission” that is implemented by meta-level programmer in order to implement the transmit meta-behaviour (explored in section 6.3.2.1). This meta-behaviour is added to the “MetaObject” that receives the message (by “addMetaTransmissionHandler” method). As a result, “MetaTransmitHandler” can have control over the message transmission from any source base-level objects to target base-level object.

The “MetaExecutionHandler” interface has a method named “handlerExecution” that is implemented by the meta-level programmer in order to implement the execute meta-behaviour (explored in section 6.3.2.1). This meta-behaviour is added to the base-level object

that receives the message (by “addMetaMarshalHandler” method). As a result, meta-level programmers can have control over how and when messages are invoked upon target base-level objects from any source base-level object.

The “MetaFactory” in Enigma instantiates an appropriate “MetaObject” object, when a base-level object is initialised. Consequently the “MetaFactory” is responsible for initializing and organizing meta-objects at the meta-level. Therefore, it is responsible for creating the meta-level and configuring it. The process of constructing the meta-level object includes the process of creation and initialization, which is handled through the ”newMetalevelFor” method of “MetaFactory”. The configuration process of meta-behaviours is handled through the “MetaHandlerClass” meta-object. The “MetaHandlerClass” meta-object encapsulates the required operations for creation and insertion of meta-behaviours. The Meta-level programmer has to specify the “MetaHandlerClass” meta-object and the type of base-objects that it applies to, shown in Figure 7.8, uses the “addMetaLevelCustomization” method of “MetaFactory”. The implementations of “MetaFactory” are facilitated by dynamic proxy in order to create meta-level, meta-level objects, explored in appendix G.

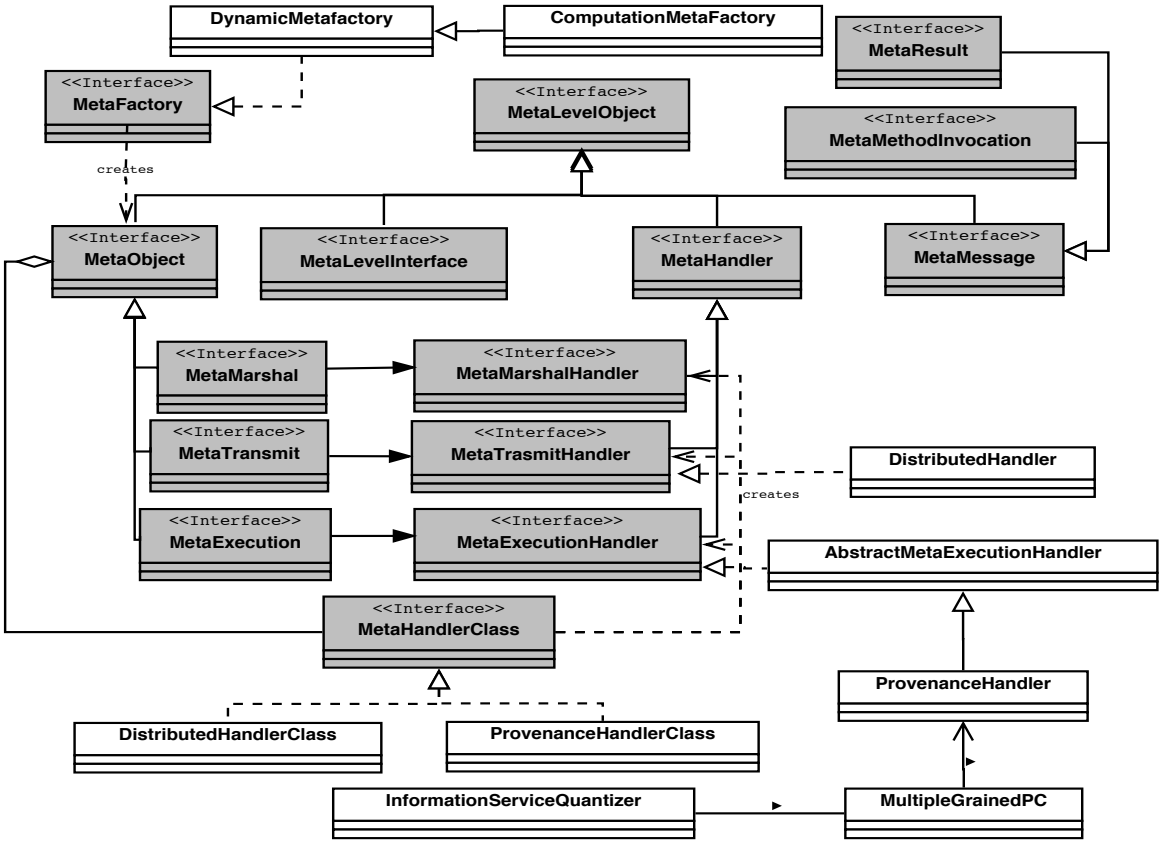


Figure 7.2. Enigma UML Class diagram.

The “MetaFactory” handles the process of creating the meta-level, creating the “MetaObject” and invoking each “MetaHandlerClass” meta-object for creation and addition of meta-behaviours. The relationship between the “MetaFactory”, “MetaHandlerClass” and meta-behaviours is shown in Figure 7.2. The “MetaFactory” is implemented in “DynamicMetaFactory” and “ComputationMetafactory”.

The “MetaFactory” interface in Enigma is responsible for constructing the meta-level for process networks. We have introduced a “Factory” interface in the PNA that is responsible for creating PN constituent components (base-level) as explored in section 3.4 and implemented in the “PNAKahnFactory” class. Both “Factory” and “MetaFactory” interfaces are used in order to construct both the base-level and meta-level. Enigma and PNA implement a factory class, named “PNAMOPFactory” that extends the PNA “Factory” interface and uses the Enigma “MetaFactory” (“DynamicMetaFactory” implementation) to construct objects in both the base-level and meta-level. This factory constructs process network components (base-objects) and also creates meta-objects whenever each base-object is created. In this approach, each base-object has corresponding meta-object at the meta-level. For example, the instantiation of an input port and reification into corresponding meta-objects in “PNAMOPFactory” is shown in Figure 7.3.

```
private MetaFactory objectMetaFactory;

public InputPort newInputPort(Process process,int index)
{
        InputPort port = new HalfChannelInputPortImpl();
        port = (InputPort)objectMetaFactory.newMetaLevelFor(port);
        return port;
}
```

Figure 7.3. Instantiation and Reification of an input port in “PNAMOPFactory” class.

7.2.1.1 The Computation meta-object

For the purposes of reification, rather than individually reify many base level objects into individual meta-objects, the Computation meta-object is a single meta-level object that aggregates the reification of a set of base-level objects. As explained in section 6.3.2.1, the concept of meta-Computation has several advantages in applying customization to all the base-objects effectively and providing runtime reconfiguration [111]. As shown in Figure 7.6,

meta-Computation is the meta-level representative of several process network objects (base-objects) [111] including process, thread of process, input port(s) and output port(s). The Computation meta-object is proposed to make the meta-level system efficient by reducing the number of meta-objects. The huge number of meta-objects and method invocations between them cause inefficiently in the meta-level system as explained in section 6.3.2.1. The Computation meta-object maintains a list of members in order to redirect method invocations to appropriate base-objects. In this model the communications between the Computation meta-objects at the meta-level are reified instead of communications of all the base-objects.

As explored in previous section, “PNAMOPFactory” uses the “DynamicMetaFactory” that is an implementation of the Enigma “MetaFactory” to a construct base-level and meta-level (objects). Enigma presents another implementation of “MetaFactory” to construct meta-level with Computation meta-objects. This “MetaFactory” named “ComputationMetafactory” the extends “DynamicMetaFactory” implementation.

In PNA, we introduce an additional factory for Computation meta-objects named “PNACompFactory” (that extends “PNAMOPFactory”) in order to reify base-objects in the form of a meta-Computation. Figure 7.4 shows how “newOutputPort” method in “PNACompFactory” creates an output port for the process network and also reifies it into the same meta-Computation as the process of that port. Therefore, the port and process are reified into a meta-Computation in meta-level with the help of “ComputationMetafactory” which is an extension of “DynamicMetaFactory”. As shown in Figure 7.4, the “PNACompFactory” uses the “ComputationMetafactory” in order to add object and its reifier into the meta-level by wrapping the invoked method and its target base-object (“TargetAction” object). Thus, Computation meta-objects are able to invoke the method upon base-object that is identified in “TargetAction”. The case-study presented in Figure 3.10 should use “PNACompFactory” instead of “PNAKahnFactory” to create both the base-level and meta-level with Computation meta-objects.

```
public OutputPort newOutputPort(Process process,int index){
        Computation computation = (Computation)process;

        OutputPort port = new HalfChannelOutputPortImpl();

        TargetAction ta = new OutputPortTargetAction(index);
        StartPointMetaObject topReifier =
        (StartPointMetaObject)((MetaLevelInterface)process).getThisMethodReifier();
        MethodReifier reifier = new ComputationReifier(ta,topReifier);
        computation.setBaseObject(ta.getTargetName(),port);

        port = (OutputPort)computationMetaFactory.newMetaLevelFor(port,reifier);
        return port;
        }
```

Figure 7.4. Creation of output port and reification of it in meta-Computation.

In this section, we have explained the implementation of Enigma in detail. We have also shown how we integrate PNA and Enigma as base-level and meta-level in an application. Therefore, we are ready to configure and implement the meta-behaviours described in chapter 6.

In the next section, we explore the implementation of provenance-collection, distribution and adaptive-granularity meta-behaviours, according to this design.

7.2.2 Meta-behaviour implementation in Enigma MOP

Recall that in section 6.3, we elaborated the design of provenance-collection, distribution and adaptive-granularity meta-behaviours that are based on the Enigma message decomposition phases as shown in Figure 7.5 and 7.6. In this section, we describe the implementation of these meta-behaviours in an Enigma meta-level application.

7.2.2.1 Provenance-collection Meta-behaviour

In this section, we present the implementation of provenance-collection meta-behaviour in the Enigma MOP designed in section 6.3. This meta-behaviour observes the behavioural and structural aspects of the workflow constituent objects. The structural aspect of this meta-behaviour is related to the structure of workflow constituent objects (including processes, channels, threads and ports). The behavioural aspect of the provenance meta-behaviour is related to communication between base-level objects in the underlying workflow system. As explained in the previous section, we can define new meta-behaviours on each message

decomposition phase through the “MetaHandler” interface. “MetaHandler” as a representation for a meta-behaviour allows customization of message decomposition phases through its interfaces (“MetaMarshalHandler”, “MetaTransmitHandler” and “MetaExecutionHandler”).

The execution phase is the phase in message decomposition for inserting provenance-collection meta-behaviour as shown in Figure 7.5 and 7.6. The customization of this meta-behaviour on the execution phase allows access to invocation information on the destination base-object. During the execution phase, provenance-collection meta-behaviour encapsulates the reified method invocation in running code that collects provenance information from the invoked method. Figure 7.5 and 7.6 show the customization of meta-behaviours upon Enigma message decomposition operating, respectively, with meta-object or meta-Computation.

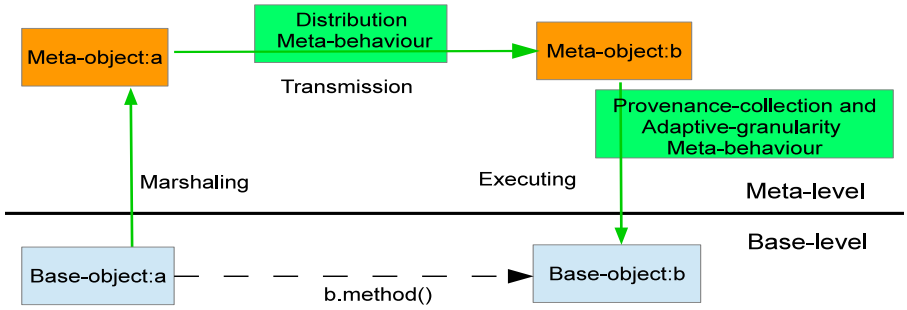


Figure 7.5. Meta-behaviours on Enigma message decomposition.

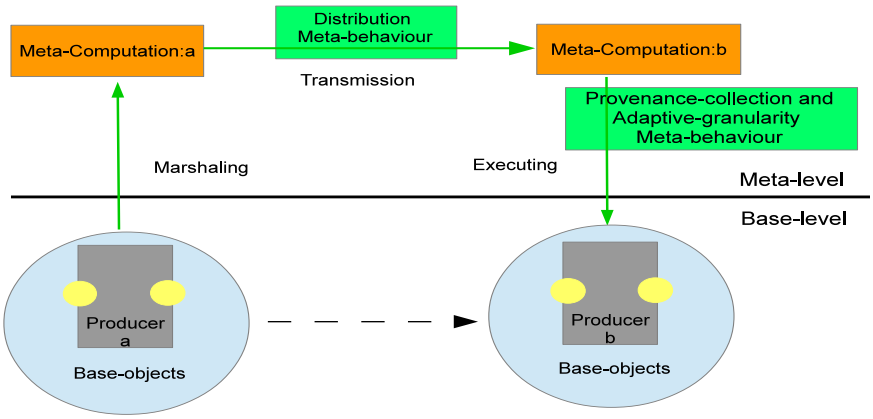


Figure 7.6. Meta-behaviours on Enigma message decomposition with meta-Computation notation.

In our implementation, provenance-collection meta-behaviour is implemented on the execution phase of Enigma message decomposition, as shown in Figure 7.5 and 7.6. This

meta-behaviour has control over messages invoked upon the target base-object from any source base-object. This allows the provenance information to be collected from the messages. We implement a “ProvenanceHandler” class for the provenance-collection meta-behaviour shown in Figure 7.7. As indicated in Figure7.2, “ProvenanceHandler” extends the implementation of the “AbstractMetaExecutionHandler” class that implements the “MetaExecutionHandler” interface. We implement the “handleExecution” method of “ProvenanceHandler” class in order to implement the provenance-collection meta-behaviour on executing meta-behaviour. This provenance-collection meta-behaviour is added to the base-level object that receives the message using “addMetaExecutionHandler” method in the “customizationMetaObject” method of “ProvenanceHandlerClass”, presented in Figure 7.8.

The “ProvenanceHandlerClass” is also added as a meta-level customization to “MetaFactory” (implemented in either “DynamicMetaFactory” or “ComputationMetaFactory” classes) in “PNAFactory” (implemented in either “PNAMOPFactory” or “PNAComFactory”), shown in Figure 7.9.

The provenance collection mechanism is managed in the “MOPMultipleGrainedPC” class, which is called in the “handleExecution” method of “ProvenanceHandler” class, shown in Figure 7.7. The “MOPMultipleGrainedPC” has a method named “methodInvocationFiltering” that takes a base-object and a method invocation upon the base-object as input parameters. In this method, provenance information is captured according to the name of the invoked methods, as shown in Figure 7.10. For example, provenance information describing the dependency of a process to its thread (agent) (a WasControlledBy dependency) is captured through the invocation of a method named “trigger” (this is a method activated process thread, it is different from the trigger method in the Main class). The “fire” method is another one that captures provenance information regarding the process dependency, “WasTriggeredBy”, through its invocations.

```
public class ProvenanceHandler extends AbstractMetaExecutionHandler
{
        public MetaResult handleExecution(Object baseObject, MetaMethodInvocation invocation)
        {
        String methodName = invocation.getMethod().getName();
        MetaResult result = getNextExecutionHandler().handleExecution(baseObject,invocation);

                if (result instanceof MetaExceptionResult)
                {
                System.out.println("Provenance: "+methodName+" threw exception");
                }else{
                        InformationServiceQuantizer isq =
                InformationServiceQuantizer.getInstance();
                if (!(isq.granularityLevel == 9))
                {
                        MOPMultipleGrainedPC CPC =MOPMultipleGrainedPC.getInstance();
                    try {
                        CPC.methodInvocationFiltering(invocation, baseObject);
                        } catch (IllegalAccessException | InvocationTargetException e) {
                                e.printStackTrace();
                        }
                }
        }

        return result;
        }
}
```

Figure 7.7. “ProvenanceHandler” class.

Prospective provenance information, which includes specification of workflow components, configurations and parameters, is captured during the workflow specification. This information includes a list of processes and networks that is collected from method invocations (of the “newProcess”, “newInputPort” or “newOutputPort” methods) before the “trigger” method in Main class (“PNATest” shown in Figure 3.10). Retrospective provenance information, which includes data derivation information and dependencies, is collected during workflow execution. This provenance information is captured from methods after the trigger method in our case study.

```
public class ProvenanceHandlerClass implements MetaHandlerClass
{
        public void customizeMetaObject(MetaLevelInterface metalevelInterface)
        {
                MetaObject metaObject = metalevelInterface.getTargetMetaObject();
                ProvenanceHandler handler = new ProvenanceHandler(metaObject);
                metaObject.addMetaExecutionHandler(handler);
        }

        public void customizeMetaObject(MetaLevelInterface metalevelInterface,Map metaInformation)
        {
                customizeMetaObject(metalevelInterface);
        }
}
```

Figure 7.8. “ProvenannceHAndleClass” class.

```
public static Factory newInstance()
        {
                MetaFactory objectMetaFactory = new DynamicMetaFactory("object");

                MetaHandlerClass ProvenanceClass = new ProvenanceHandlerClass();
                objectMetaFactory.addMetaLevelCustomization(Object.class,ProvenanceClass);

                Factory factory = new PNAMOPFactory(objectMetaFactory);
                return factory;
        }
```

Figure 7.9. "newInstance" method of "PNAFactory" class.

The provenance collection mechanism is managed in the "MOPMultipleGrainedPC" class, shown in Figure 7.7. The "MOPMultipleGrainedPC" class is a singleton class, which is responsible for collecting and storing provenance elements (such as provenance dependencies) according to a MoP that is compatible with OPM format. This class constructs provenance dependency determined by the MoP from invoked methods, as shown below. The "methodInvocationFiltering" method of "MOPMultipleGrainedPC" class filters invoked methods and delegates the methods to the corresponding OPM dependencies as presented in Figure 7.10 and explained below.

- *WasControlledBy()* is collected when the "trigger" method is reified. The "trigger" is a method in "Process Thread" to activate a controller (Agent) for a process.
- *WasTriggeredBy()* is collected when the "fire" method in the process is reified. The "fire" method is called by Agent ("ProcessThreadImpl" class) to fire a process.
- *WasGeneratedBy()* and WasDerivedFrom() are collected when the "getData" method in process is reified. The "getData" method is called by Agent ("Process Thread") after fire method execution.
- *Used()* is collected when the "preFire" method in process is reified. The "preFire" method is used by the Agent ("Process Thread") to prepare the process for execution, which includes taking input data from input channel.
- *I-WasDerivedFrom()* intermediate dependency is collected each time the "fire()" method in the process is executed (invoked). The created data is called intermediate data. This dependency is captured after the "fire()" method because it is the only method inside the process that creates data.

```
        if ((method.getMethod().getName().equalsIgnoreCase("trigger"))) {
                                createWCB(invocation, baseObject);
        }if ((method.getMethod().getName().equalsIgnoreCase("preFire"))) {
                                createUsed(invocation, baseObject);
        }if ((method.getMethod().getName().equalsIgnoreCase("fire"))) {
                if (!(((AbstractProcess) baseObject).getPreviousProcess() == null)) {
                                createWTB(invocation, baseObject);
                }
                                createI-WDF(invocation, baseObject);
        }if (invocation.getMethodName().startsWith("getData")) {
                                createWGB(invocation, baseObject);
                                createWDF(invocation, baseObject);
        }
```

Figure 7.10. Methods for constructing OPM dependencies in “MOPMultipleGrainedPC” class.

In our workflow case study, a “trigger” method is used to start and activate the execution of a thread for a process. As this method is invoked during the workflow execution, this method invocation is reified into the meta-level. The method invocation is delegated to the “methodInvocationFiltering” method of the “MOPMultipleGrainedPC” class that filters invoked methods, as shown in Figure 7.10. The “methodInvocationFiltering” method delegates the “trigger” methods to the “createWCB” method in order to construct a “WasControlledBy” dependency. The “createWCB” method contains two entities of process and process thread that are used to construct and represent the “WasControlledBy” dependency.

Now, we are going to discuss aspects of meta-Computation (introduced as the Computation meta-object in section 7.2.1.1) related to our provenance collection mechanism. Provenance collection mechanisms at the reflection layer (meta-level), either in PNA with or without the notion of meta-Computation, mostly deals with the process agent in order to collect provenance by trapping the invoked methods. The process agents are the same in both PNA with and without the notion of meta-Computation. Therefore, PNA with and without the notion of meta-Computation has the similar functionality in terms of provenance collection mechanism. However, the implementation and design principles of provenance implementation in a PNA with and without the notion of meta-Computation are substantially different.

Because most selected method invocations for provenance collection purposes at the meta-level (Enigma) originate from the process agent, which is a Java thread that runs and applies the MoC semantics to processes in workflows. As such, our provenance collection mechanism implemented on Enigma can operate similarly on a PNA with and without the notion of meta-

Computation. For example, method invocations, such as “fire” methods originated from the process thread (agent), which is eventually delegated to the actual process. In both approaches, reification of “fire” methods are used in a similar manner to collect provenance in meta-level.

The main advantage that Enigma with a notion of meta-Computation can offer (compared to not using it) is the ability to collect “WasTriggeredBy” and “WasDerivedFrom” dependencies in one pass. Meta-Computation in Enigma is facilitated by a data structure contained in processes containing information about input ports and output ports, initial parameters, the name of previous process, and, optionally, intermediate produced data. Therefore, in case of “WasDerivedFrom”, it would be possible to access tokens in input ports and output ports to construct the dependency. Input and output ports contain tokens that come from output channels and get into input channels in a process. These two dependencies are usually inferred from generated “Used” and “WasGeneratedBy” dependencies in the OPM file, as explained in section 4.5.2.

7.2.2.2 Distribution Meta-behaviour

Distribution meta-behaviour is implemented in Enigma to handle remote method invocations. As explained in section 6.3.2.2, the distribution meta-behaviour aims to make the execution of workflow constituent objects on distributed environments transparent from base-level application’s point of view. The distribution meta-behaviour is an important part of Enigma design but we do not use it in our case study in this thesis. This meta-behaviour modifies where the reified invocation is sent. It intercepts a method invocation and manages the process of transmitting method invocations and results to a possibly remote host where the target base-object of method invocation is located.

Enigma [111] defines the distribution meta-behaviour in the form of a remote transmission meta-behaviour acting as a proxy to a server object of a remote method invocation (where the target object is located). As shown in 7.5 and 7.6, Enigma implements distribution meta-behaviour in the “handleTransmission” method of “MetaTransmissionHandler”. In this model, all method invocations are propagated to the server and new transmission meta-behaviours are added to remote servers (implemented in the “setNextTransmissionHandler”

method). The “addMetaTransmissionHandler” method of the target meta-object adds “RemoteTransmissionHandler” to the meta-level.

7.2.2.3 Adaptive-granularity Meta-behaviour

The adaptive-granularity meta-behaviour and the way it works are the key points of this study. The way the provenance collection mechanism adaptively adjusts to collect an appropriate level of provenance granularity based on changes in the requested level of provenance is explored in this section.

We implement a singleton class named “InformationServiceQuantizer” to get information from the Information Server. The “InformationServiceQuantizer” (shown in Figure 7.2) quantizes the requested level of granularity to fine-, medium-, coarse-grained provenance and no-provenance. For example, if a user selects an adaptive provenance policy in the Information Service, the Information Service decides the level of granularity based on all the environmental information that it has including resources’ price, budget, deadline and system performance and channels’ condition. It then informs the adaptive provenance collection mechanism (through the “InformationServiceQuantizer” class) about the level of granularity that should be collected.

The “InformationServiceQuantizer” class is used in provenance collector methods that collect OPM dependencies in the “MOPMultipleGrainedPC” class, such as “createWGB” dependency in Figure 7.11. The methods of “MOPMultipleGrainedPC” as shown in Figure 7.10 (including CreateWCB(), createUsed(), createWTB(), createI-WDF(), createWDF() and createWGB()) aim to construct OPM dependencies according to presented MoP in section 4.5. These methods (such as the “createWGB” dependency in Figure 7.11) have separate implementations for different levels of provenance granularity to construct related dependencies (such as the WasGeneratedBy dependency in Figure 7.11). The implementations follow the MoP pattern presented in chapter 4.

Our implementations of adaptive provenance collection support (data-oriented) multiple-granularity MoP for fine, medium and coarse-grained provenance were presented in section 4.5.2 as shown in appendix A, B and C respectively. Appendix D shows the coarse-grained

provenance based on a coarse-grained MoP presented in section 4.5.1 that represents all OPM dependencies in it. Appendix E shows multiple-granularity provenance that contains provenance information in three levels of provenance granularity (fine, medium and coarse-grained). The collected provenance information in appendix E has three accounts (account 1, 2 and 3). It was generated from a situation in which the Information Service requests to change the level of granularity twice. Thus the dependencies and artifacts are collected under different accounts and granularity. Suppose that firstly it collects fine-grained provenance with account id = 3, then it collects medium-grained provenance with account id =2 and finally coarse-grained provenance with account id =1.

```
private void createWGB(MetaMethodInvocation invocation,Object target) {

  if (!(isq.granularityLevel == "NoProv")) {
        if ((isq.getGranularityLevel() == "CoarseProv")) { //coarse-grained provenance
        if ((isq.getGranularityLevel() == "MediumProv")) { //medium-grained provenance
        if ((isq.getGranularityLevel() == "FineProv")) {  //fine-grained provenance
  }
```

Figure 7.11. The createWGB method decides about the level of provenance granularity.

In this section, we have shown how the meta-behaviours in chapter 6 are implemented in an Enigma meta-level application. In the next section, we present an implementation of our provenance collection design based on an AOP software architecture, as explained in chapter 6.

In the following sections aspects of provenance-collection and adaptive-granularity are explored.

7.3 AOP oriented adaptive provenance collection mechanism

The design principles of adaptive provenance collection mechanisms based on AOP software techniques are explained in section 6.4. In that design, a PNA is considered as an independent functional application that is integrated with the AspectJ framework to construct an AOP oriented provenance collection mechanism. Two aspects - provenance-collection and adaptive-granularity - are defined aiming to express the idea of provenance collection and multiple-granularity provenance.

In this section, we show how PNA (functional application) works independently from the aspect application that is responsible for collecting provenance. As shown in Figure 7.12, our case study uses a similar process network to the MOP case study shown in Figure 7.1 as an underlying workflow system implemented in a PNA framework. This similarity in the process network case study provides a chance of compare AOP and MOP oriented mechanisma fairly. In this case study, a "PNAKahnFactory" factory class is used to independently construct a PNA framework and creates process network constituent components, as explained in section 3.4. In the MOP oriented case study, the Factory class was responsible for construction and initialization of the meta-level. Figure 7.12 shows the integration of our process network case study with the AspectJ framework to construct our case study.

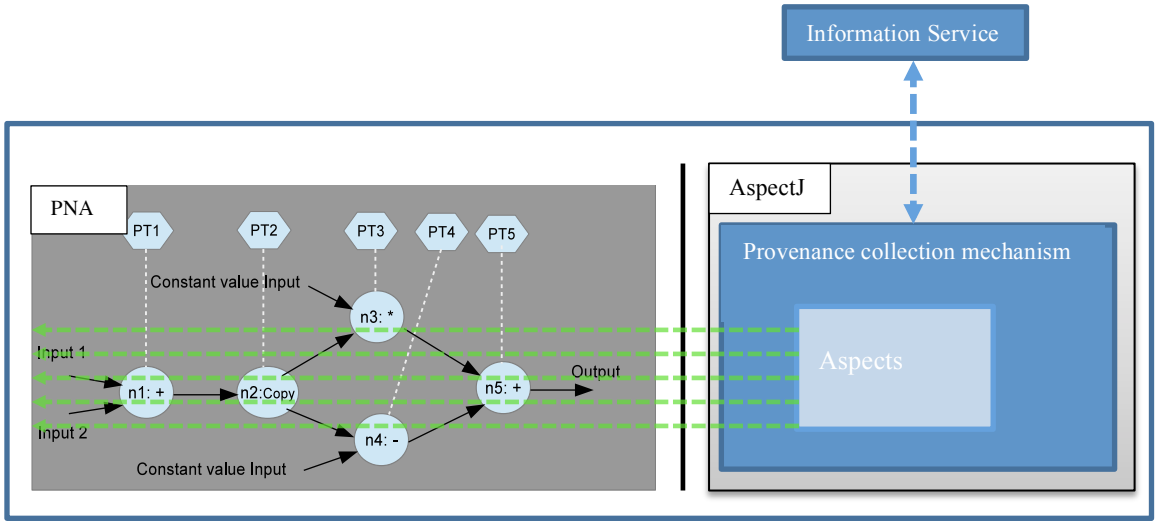


Figure 7.12. Case-study architecture: Aspect viewpoint.

7.3.1 Implementation of the Provenance Aspect

The provenance aspect aims to collect provenance information and store it according to the defined MoP in section 4.5, which is compatible with OPM representation. As explained in section 6.4 and shown in Figure 7.13, the defined pointcuts in the aspect application applied to components in PNA application would determine where an aspect aims to collect provenance information. The number of pointcuts and the point in program to which they refer is the required based on the defined MoP. The advice method assigned to the pointcut is responsible for collecting requested provenance information and presenting in the requested format in the MoP explored in section 4.5.

We can collect both prospective and retrospective provenance (explained in section 5.1.4) during the workflow lifecycle in a way the similar to the MOP oriented mechanism, explained in section 6.1.2. The MoP defined in section 4.5 shows some prospective provenance. The MoP has a form of provenance specification that includes a list of processes, artifacts, accounts and agents. The MoP represents the retrospective provenance in the form of OPM dependencies, including WasGeneratedBy(), Used(), WasTriggeredBy(), WasControlledBy(), WasDerivedFrom().

In terms of implementation, as shown in Figure 7.13, we implement an abstract aspect class named "AbstractProvenanceModel", which identifies the required pointcuts. In addition, there is an implementation –the "OPMMultipleGranularityPC" aspect class- for this abstract class that extends the pointcuts and implements their advice method to collect and store provenance information in OPM format according to the required granularity.

In our implementation of the "OPMMultipleGranularityPC" aspect, we implement a pointcut for each OPM dependency such as WasGeneratedBy(), Used(), WasTriggeredBy(), WasControlledBy(), WasDerivedFrom() pointcuts. Each pointcut is assigned to specific method in PNA, as explored below. For example, there is a WasGeneratedBy (WGB) pointcut. The pointcut refers to the getOutputPort() method in processes across the PNA framework. Figure 7.14 shows our design for the pointcut of OPM dependencies.

Different ways of constructing OPM dependencies are explained below:

- The *WasGeneratedBy* dependency is constructed after the execution of "getOutputPort" method in the process. This dependency is constructed when artifacts are delivered to channels.
- The *Used* dependency is constructed before the execution of the "getInputPort" method in the process. This dependency is constructed when artifacts are delivered to processes.
- The *WasTriggeredBy* dependency is constructed before the execution of "fire" method in process. This dependency is constructed when the process is activated by previous process(es).
- The *WasDerivedFrom* dependency is constructed during the execution of the "fire" method in the process. This dependency is constructed when artifacts are delivered to process(es).
- The *WasControlledBy* dependency is constructed after the creation of the "Process Thread" object, which is an Agent for a process.

- The *I-WasDerivedFrom* intermediate dependency is constructed after each execution of the "fire()" method in Process. The created data is intermediate data. This dependency is captured after the "fire()" method because it is the only method inside process that creates a data.

In our workflow case study, a process creates artifacts (data) and puts them into its output port with the "getOutputPort" method. As shown in Figure 7.13 and Figure 7.14, our provenance collection mechanism has a WasGeneratedBy (WGB) pointcut that is implemented in the "OPMMultipleGranularityPC" aspect which refers to the "getOutputPort" method. Therefore, the invocation of this method activates the WasGeneratedBy (WGB) pointcut, which enables the advice method of this pointcut to collect provenance information. The advice method collects and constructs provenance information according to the MoP for that level of granularity. Our MoP determines the information that should be collected in the form of pointcuts that are defined according to each level of provenance granularity. It also determines the way the collected provenance should be represented, which in this case is a format compatible with the OPM MoP, for example for coarse-grained provenance, the MoP includes a list of agents, processes, artifacts and dependencies. Our MoP and its constituent objects were explored in section 4.5.

In the case study shown in Figure 7.13, the WGB pointcut defines three advice logics implementing the collection and presentation of provenance information. Advices implement the way provenance should be represented in each pointcut for levels of provenance (coarse, medium and fine-grained). In this case shown in Figure 7.13, the invocation on "getOutputPort" method activates the WGB pointcut and its advice logic. If the requested provenance granularity by the Information Service is coarse-grained, this advice as the main component of provenance collection mechanism firstly receives the artifact (data) and process. Then, the advice adds them in a list of processes and artifact (if they have not been added before). Following that the collection mechanism constructs the dependency between the artifact and the process that was generated by that artifact ("WasGeneratedBy" dependency).

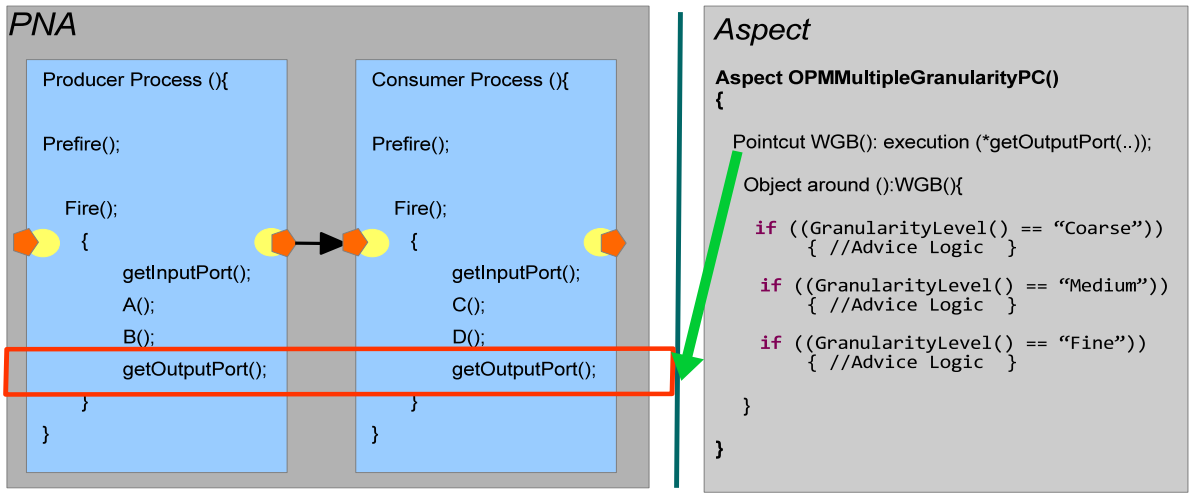


Figure 7.13. The AOP case study.

```
pointcut WasGeneratedBy(): execution(au.edu.adelaide.kahn.pn.AbstractProcess.getOutputPort(..));

pointcut Used():execution (* au.edu.adelaide.kahn.pn. AbstractProcess.getInputPort (..))

pointcut wasDeriviedFrom():execution (* au.edu.adelaide.kahn.pn.Process.fire(..))
                          && target(au.edu.adelaide.kahn.pn.AbstractProcess);

pointcut wasTriggerBy():execution (* au.edu.adelaide.kahn.pn.Process.fire(..))
                        && target(au.edu.adelaide.kahn.pn.AbstractProcess);

pointcut wasControlledBy():execution (au.edu.adelaide.pagis.system.ProcessThreadImpl.new(..)) ;

pointcut I-wasDeriviedFrom():execution (* au.edu.adelaide.kahn.pn.Process.fire(..));
```

Figure 7.14. Pointcuts in AspectJ collecting provenance information for OPM dependencies.

In our workflow case study, a similar scenario occurs for the "Used" dependency in input ports of processes as explained earlier and shown in Figure 7.14. As shown in the WasGeneratedBy example (the situation is the same for "Used"), whenever a pointcut in an aspect is set, it has access to that specific point in the program; however, a provenance collection mechanism needs to have access to more than one point (or method) in the source code to collect information for the "WasDerivedFrom" and "WasTriggeredBy" OPM dependencies. It is not possible for a pointcut to access more than one point or method. For example, a provenance collection mechanism aiming to construct a "WasDerivedFrom" dependency needs to have access to artifacts in an output channel and it also needs to know the corresponding artifact in the input channel (If we are going to collect provenance dependency at runtime and in one pass). This is the reason that we implemented the "WasDerivedFrom" dependency's pointcut in the Process entity instead of input and output channels. The "WasDerivedFrom" and "WasTriggeredBy" pointcuts refer to the "fire"

method of the process entity. In the process entity, a list of parameters are stored which make it possible to access the input value from input channel at the same time as accessing the output value. The scenario is similar for "WasTriggeredBy" where it is necessary to know about the process that triggers the current one, which is recorded in process entity.

We use the same MoP (explored in section 4.5.1) for both AOP and MOP oriented provenance collection mechanisms. Therefore, both mechanisms use the same representation of provenance granularity, which causes both mechanisms collect similar provenance information. We show the collected provenance from our case study in appendix A, B, C and D. Coarse-grained provenance information is presented in appendix C and D based on the coarse-grained MoP presented in section 4.5.1 and 4.5.2, respectively. The medium-grained provenance level is identified with specific account id=2 shown in appendix B based on the presented medium-grained MoP in section 4.5.1 and 4.5.2. The AOP provenance collection collects at the fine-grained provenance level in account id =3 shown in appendix A based on the presented fine-grained MoP in section 4.5.1. We will pose a discussion in section 7.3.1 regarding fine-grained provenance. In the next section, the adaptability of this provenance collection mechanism is investigated.

7.3.2 Implementation of Adaptive Aspect

In this section, we explore how adaptation works in an AOP oriented provenance collection mechanism in order to adjust the level of provenance granularity. As explained in section 7.2.2.3, the adaptive provenance collection mechanism changes the level of provenance granularity based on the level specified by the Information Service. The "InformationServiceQuantizer" class takes a requested level of granularity from the Information Server and informs the provenance collection mechanism about the level of granularity including fine-, medium-, coarse-grained provenance and no-provenance. The implementation of this aspect is similar that of adaptive-granularity meta-behaviour.

As explained we are using the AspectJ framework, which is facilitated by a compile-time injection of code. That injected code is written to the required level of provenance collection. In the implemented aspect, all the pointcuts are always set and active on these aspects. So

whenever program reaches the specified point in a program, the advice defined for the pointcut is ready to its perform particular action. In this situation, there are three separate implementations for different level of provenance granularity in each advice shown in Figure 7.13. Therefore, an advice firstly checks the defined level of provenance (similar to Figure 7.11) and activates the implementation for that level of provenance. For example, suppose that the Information Server determines the collection of fine-grained provenance information. In order to collect WGB OPM dependency, one pointcut is assigned for WGB dependency in the aspect class as shown in Figure 7.13. Whenever program execution reaches the getOutputPort() method, the WGB pointcut runs the advice method. In the advice method only the advice logic would collect provenance information related to the WGB dependency defined in the fine-granularity MoP.

In this section we explained the implementation of provenance-collection and adaptive-granularity aspects of the AOP oriented provenance collection mechanism. We evaluate and compare the implementation of the AOP and MOP oriented provenance collection mechanism in the next section.

7.4 Evaluation and Comparison

Table 7.1. Design dimensions of our MOP and AOP oriented adaptive provenance collection.

Software technique	**Workflow Orientation**	**Provenance collection phases**	**Level of abstract**	**Prospective (P) Retrospective (R)**	**Granularity**	**Level of access to workflow information**	**MoP**	**Coupling strategy**	**Type of instrumentation**	**Storage**
MOP & AOP	Scientific	Specification & Execution	Data & Process	P: XML R: XML	Multiple-granularity	Workflow-level & activity-level	OPM Compliant	Loosely-coupled	Non-functional concerns	XML

As shown in Table 7.1, the implementation of adaptive provenance collection based on MOP and AOP software techniques satisfy the desired design dimensions explored in section 6.1.2.

In section 6.1.2, we explored adaptability and multiple-granularity provenance features in terms of a number of design dimensions such as accessibility of detailed information, architecture layers of provenance systems and type of instrumentation. The key point of understanding from section 6.1.2 is that the desirable attribute of adaptability appears to be

achievable through using a non-functional concern instrumentation. The multiple-granularity feature requires a particular level of access to all levels of workflow and system information that is not supported in any of the previous provenance systems.

Now, we discuss Table 7.1 in more detail. Our implementation of MOP and AOP oriented provenance collection mechanism offers some valuable properties including: multi-granularity provenance, non-functional concerns and access to both workflow and activity level information (explored in following paragraphs). These properties are explored as desired design dimensions for the adaptive provenance collection mechanism in section 6.1.2. Our provenance collection mechanism addresses design dimensions that the provenance collection mechanisms shown in Table 5.1 do not address, such as multi-granularity provenance, non-functional concerns and access to both workflow and activity level information.

Multi-granularity provenance is represented in a multi-granularity MoP as presented in section 6.5. The MoP represents what provenance information and dependencies should be collected by the MOP and AOP oriented adaptive provenance collection mechanisms. The upcoming changes in the underlying environment and the requirements of the user and system trigger corresponding changes the level of provenance granularity. The collected fine-, medium- and coarse-grained provenance information is shown in appendix A, B and C respectively. The collected multiple-granularity provenance information is shown in appendix E.

Provenance design which leverages a separation of concerns offers the potential for a clear and flexible design. The provenance collection mechanisms used non-functional concerns instrumentation that enables the implementation of provenance collection mechanism in a separate application that supervises workflow. The MOP and AOP are flexible and extensible approaches that do not interfere with the primary workflow execution. They have not yet been deployed (for the purposes of provenance collection) outside of this work. These approaches provide access to both the workflow and activity level of information required for provenance collection, as shown in the level of access to workflow information in Table 7.1.

There are currently limitations to these mechanisms, for example, neither MOP or AOP can access the OS-level of workflow information. Accessing workflow information at the OS-

level enables access to all the method calls and system calls in the system. It can be viewed as a special case of the fine-grained provenance collection mechanism, explored in section 7.4.1. The MOP mechanism of collecting provenance sits on top of workflow system while the AOP mechanism is cross-cut the workflow layer. The provenance collection mechanisms implemented in this chapter enable collection of provenance in both workflow specification and execution phases. Therefore, as indicated in the prospective and retrospective section of Table 7.1, they are firstly capable of collecting prospective and retrospective provenance information and represent the information in XML format.

The implemented MOP and AOP oriented adaptive provenance collection mechanism are loosely-coupled to a workflow system, as presented in the coupling strategy section of Table 7.1. The provenance information collected by these mechanisms is modelled in a format compatible with OPM, which makes it interoperable (though not necessary fully interoperable as explained in section 4.4.).

The implementation of an adaptive provenance collection based on MOP and AOP software techniques collects medium- and coarse-grained provenance based on multi-granularity MoP. The MoP and AOP oriented adaptive provenance collection mechanisms are capable of collecting data-oriented fine-grained provenance information represented in a data-oriented multiple-granularity MoP presented in section 4.5.2. However, there are some arguments and issues regarding collection fine-grained provenance information that we address in the next section.

We used the AspectJ framework to implement our AOP oriented provenance collection mechanism. AspectJ, as explored in section 2.4.2.2, is a well-defined aspect-oriented framework that easily integrated with our process network case study. It handles all the aspect weaving automatically. Moreover, it has a plug-in (AJDT) that makes it possible to develop an AOP program within the IBM Eclipse IDE. The implementation of the AOP oriented provenance collection mechanism is easier than the MOP mechanism, because Enigma has to manage all the mechanisms of reflection including meta-level construction, instantiation of meta-objects and message decomposition. Further the desired meta-behaviours including provenance-collection and adaptive-granularity must be introduced through Enigma mechanisms. Apart from the ease of implementation in AOP mechanisms, we found that the

AOP mechanism has a clearer design approach which makes the design of provenance-collection and adaptive-granularity concerns easier to understand and express.

7.4.1 Comments on the implementation of fine-grained provenance

Several MoPs, such as OPM, OPMW, and PROV, demonstrate the provenance dependencies between and among Processes (Activities), Artifacts (Entities) and Agents. The fine-grained MoP presented in section 4.5.1 and 4.5.2 contains some extra provenance information expressed in a form of intra-process dependency and intermediate data. One of the issues in capturing intra-process dependencies is identifying a structure for elements inside processes to establish relationships between them. The structure for elements is a recognizable unit of the applications, for example a Process entity in OPM.

In order to tackle the identification of a structure for recognizable units of applications in our implementation, we assume a process in workflow system could be viewed as a composite process containing several processes (sub-processes) and each process contains a number of methods. Methods and sub-processes are considered as recognizable units (elements) inside processes. Therefore, fine-grained MoP should capture the dependencies caused by interaction between all the methods and between sub-processes in a process. In this implementation, sub-processes are considered at the level of process and the dependency is modelled in OPM as indicated by the intra-dependency in section 4.5. But capturing dependencies between method invocations is still an issue, because the collection mechanism needs to capture all the invoked methods.

As explained in section 2.1.1, the definition of fine-grained provenance of interest to the database community [72, 75] captures the provenance information from all computational steps (method invocations) [77, 183] regardless of the structure and components of the system that provenance collection mechanism works on. However, our fine-grained provenance is defined with consideration of the structure and components of workflow systems with a distinction maintained between inter-process and intra-process provenance. Therefore, it is important for our mechanisms to differentiate a list of method invocations for capturing provenance from all the invoked methods, because the method invocations determine the

dependencies between methods (caller methods would invoke callee methods, established dependencies between callee and caller).

In many workflow systems, which are component-based or actor-oriented, the interface of a component specifies a pattern of methods. For example, the process component in our implementation has a process interface including "preFire", "fire", "getInputPort" and "getOutputPort" methods, as shown in Figure 7.3. The provenance information (including dependency and intermediate data) of these method invocations is captured in our implementation. However, there are a number of other methods *inside* the process (including A(), B(), C() and D() shown in Figure 7.3) that are not considered in our provenance collection mechanisms. The methods defined in the Process interface are considered for capturing the intermediate (intra-process) dependencies in both of our implementations in this thesis. Our provenance collection mechanisms are not designed for capturing intermediate dependencies between all the invoked methods.

In AOP approach, pointcuts in aspects use a method name to refer to a specific method. Pointcuts refer to methods using a name-convention, for example, they capture the "getOutputPort()" method in all processes in Figure 7.3. The methods defined in the Process interface are only repeated in all processes such as "prefire" and "fire" methods: consequently, they can identified by their name, because it is known and predictable. However, there is no consistent name-convention that encompasses *all* methods inside processes; and different methods may be invoked in different processes. Therefore, AOP is not able to refer to all methods (in processes), because those methods do not follow a specific pattern for their method (represented with the same name in processes). For example as shown in Figure 7.13, method A() and B() are invoked in the producer process while method C() and D() are invoked in consumer process. In addition, other method patterns may exist in other processes. As a result, AOP in its basic form is not capable of easily capturing intermediate methods, and consequently it is not able to construct all intermediate dependencies between them. We note that it is possible in AOP framework using wildcards to catch all the method invocations of all objects, but it does not help our provenance collection mechanism to construct provenance dependencies while it considers the structure of workflow in producing provenance information (based on our definition). The use of wild cards captures

all method calls without capturing knowledge of how they fit into the structure of the application.

In the MoP approach, method invocations in PNA are reified into the Enigma meta-level application. However, the only method invocations that are reified are the ones that belong to the PNA process thread, such as the “prefire”, “fire” and “postfire” methods (refer to Figure 3.8). The process thread delegates the execution of these methods inside the process nodes, for example, the “fire” method in process thread is delegated to the “fire” method in the process node (shown in Figure 7.3). Following that the invoked methods in the PNA are reified into the meta-level.

However, methods in the process nodes are not visible from the meta-level. This includes methods inside the fire method as shown in Figure 7.3, such as methods A(), B(), C() and D(). These methods are not defined in the process interface and they are not invoked directly from process thread. Therefore, our MOP oriented provenance collection mechanism that operates from the meta-level is not able to capture intermediate methods, and consequently construct intermediate dependencies between them. A provenance system that is going to collect all intermediate methods, data and dependencies (as fine-grained provenance) requires a provenance collection mechanism that has a access to information in the operating system, application, or the compiler level [171, 183]. As explained in section 5.1.12, there are other mechanisms providing access to all the method calls or systems calls such as the Javassist [207, 208], implemented mechanism in SPADE [171] and Data Tracker [77].

8 SUMMARY AND CONCLUSIONS AND FUTURE WORK

8.1 Summary

Scientific workflow systems provide scientists with a framework that can greatly improve the organisation, repeatability and documentation of experiments. SWfMSs manage and represent complex, heterogeneous and distributed scientific computations. Scientific workflow systems are supported by provenance systems, which enable scientists to validate their hypotheses and verify results. Provenance is shared with results of experiments to improve interpretation and understanding of final results by examining the chain of reasoning and sequence of steps that led to the final results. Provenance enables scientific workflow systems to reproduce results, retrace experiments and repeat experiments.

This thesis has studied and surveyed provenance collection mechanisms and models of provenance in order to design adaptive provenance collection mechanisms. Our adaptive mechanism is capable of collecting at a particular level of provenance granularity in response to changes that workflows face in the computing environments in which they operate. In doing so, this work provides an understanding of separation of concerns as a guiding principle behind the design and development of adaptive provenance collection mechanisms.

In this work we examined the Model of Computation (MoC) in scientific workflow systems. The Synchronous Dataflow (SDF) and Process Network (PN) MoC are explored in our simple dataflow model and in the context of the Kepler workflow system. We justified the selection of PN as an underlying semantics for scientific workflow systems. We developed a Process Network Application (PNA) to express the notion of scientific workflow system. We use this application to implement our workflow case study.

The development of provenance collection mechanisms reveals the importance of making explicit the concept of Model of Provenance (MoP), which determines what should be considered as provenance information and how it should be represented. We elaborated the concept of MoP in a simple MoP case study. To address the adaptive requirement of provenance collection mechanisms, we introduced a MoP, which is capable of collecting and representing provenance information at multiple levels of granularity. The MoP was designed

for compatibility with the Open Provenance Model (OPM) to provide interoperability with other provenance systems.

Our aims of developing provenance collection mechanisms motivated us to identify and define a set of design dimensions for provenance collection mechanisms. After this, we surveyed a set of scientific workflow projects based on our design dimensions with an emphasis on provenance collection mechanisms. This survey provides an understanding of primary design issues for provenance collection mechanisms along with a set of desirable design dimensions.

We presented a clear design for provenance collection in a scientific workflow architecture. Our provenance architecture benefits from separation of concerns that untangles the adaptive-granularity and provenance-collection concerns, which results in an adaptive provenance collection mechanism with a simple design with minimal impact on the core structure of the SWfMS. Our adaptive provenance collection mechanisms use reflection (MetaObject Protocol (MOP)) and Aspect-Oriented Programming (AOP) software development techniques as two different ways to realize this separation of concerns.

We proposed two MOP and AOP oriented designs for adaptive provenance collection mechanisms, considering provenance-collection and adaptive-granularity as key elements of design. The MOP oriented adaptive provenance collection mechanism follows the design principles of the Enigma meta-level application. Similarly, the AOP oriented adaptive provenance collection mechanism is designed and implemented using AspectJ. The implementation of both MOP and AOP oriented adaptive provenance collection mechanisms are integrated with our scientific workflow case study, implemented in PNA.

8.2 Contributions

The following contributions are presented in this thesis:

1. *The primary contribution of the thesis is the definition of an architecture for adaptive multiple granularity provenance collection, supported by the implementation of a simplified case study (Chapter 4 and 6).* We primarily aim to design adaptive provenance

collection mechanisms capable of collecting various levels of provenance granularity. Therefore, we proposed two distinct views of adaptive workflow architectures separating function and behaviours in workflow architectures. These views provide the adaptive-granularity and provenance-collection concerns in adaptive provenance collection mechanisms (chapter 6). In addition, we proposed a multiple-granularity MoP for our adaptive provenance collection mechanisms to determine what provenance information should be collected and how this information is represented (section 4.5).

2. *Design and implementation of an adaptive-granularity provenance architecture based on the principle of separation of functional and behavioural concerns, using two ways of realizing that separation: MetaObject Protocol and Aspect-Oriented programming (Chapter 6 and 7).* Our adaptive provenance architecture employs separation of concerns as its guiding principle behind its design. We design, implement and compare our adaptive provenance collection mechanisms according MOP and AOP software techniques to employ and express the separation of adaptive-granularity and provenance-collection concerns (section 6.3 and 6.4).

3. *A study of the various Models of Provenance (MoP), leading to a new view of multiple-granularity provenance (Chapter 4).* We made explicit the concept of a MoP by exploring a case study and surveying this concept as explored in the literature (both implicitly and explicitly) (section 4.2, 4.3 and 4.4). We also made a connection between a MoP and level of provenance granularity. This connection justifies the design of a MoP for various levels of provenance granularity. Our MoP enabled our adaptive provenance collection mechanisms to collect provenance information at various level of granularity according to (what is determined as provenance information in) the MoP; and represent this information based on (representation format identified in) the MoP.

4. *Strong OPM compliance in our adaptive multiple-granularity MoP, designed for interoperability, and demonstrated by a case-study implementation (Chapter 7).* We provide two implementations of adaptive provenance collection mechanisms that are capable of selecting a particular level of granularity from the multiple-granularity MoP. As a result of selecting a level of granularity, both adaptive provenance collection mechanisms collect and represent provenance as standardized in the selected level of the MoP. The implementations of our both mechanisms are capable of representing provenance in a format compatible with the OPM, because their MoP is designed according to semantics and rules of OPM MoP. Using OPM MoP enables interoperability

of our provenance system. The implementations are satisfied the scenario for provenance collection in a scientific workflow system explored in section 7.1. We explore the design and implementation this MoP in our case-studies in chapter 6 and 7.

5. *Identifying and defining design dimensions for provenance collection mechanisms, in order to survey existing provenance collection mechanisms in scientific workflow systems (Chapter 5).* We identified and defined a set of design dimensions for provenance collection mechanisms working on scientific workflow systems, and used them to survey and evaluate a number of existing provenance collection mechanisms.

6. *Refinement of an earlier meta-object protocol based Process Network implementation, into a Process Network Application (PNA), representing a Process Network, as a representative scientific workflow system for a case study Chapter 3).* We developed a Process Network Application (PNA) in section 3.4. PNA provide an architecture utilized by a PN Model of Computation as an underlying semantics of execution to express the notion of scientific workflow systems.

8.3 Future work

We have employed a non-functional concern mechanism of instrumentation in our design and implementation. The mechanisms using MOP and AOP software techniques are not able to collect low level provenance information by collecting all intermediate methods, data and dependencies in a fine-grained provenance view, as explored in section 7.4.1. This thesis outlines a number of solutions for dealing with this issue, including employing alternative mechanisms of instrumentation enabling access to all the method calls or systems calls. Compiler-based instrumentation [77, 171, 207, 208] will potentially be an important direction as this work continues.

We have used AspectJ as an AOP implementation tool that is capable of compile-time weaving. Runtime weaving implementation tools for AOP including Spring AOP, JAC and JBoss would further improve the adaptability of the AOP mechanism. Using runtime weaving tools also provide a framework to compare them with our AOP oriented mechanism in terms of ease of implementation and functionality. Noting that, coupling AspectJ AOP and reflective programming [133, 134, 209] would be another interesting future direction for this

research. This coupling can benefit from the dynamic proxy of our reflective mechanism (providing runtime weaving), and the fine-grained pointcuts declaration proposed by AspectJ.

We have not yet analysed our MOP and AOP approaches from the aspect of performance. Analysing these approaches in terms of time and cost of execution and integration provide a framework to compare these approaches in a more comprehensive way. In addition, to compare these approaches with the coupling approach of AOP and MOP.

We are interested in extending the design of our multiple-granularity MoP to also be compatible with ontology based MoPs including PROV and OPMW.

We have used our workflow case-study in this thesis. Applying our adaptive provenance collection mechanism to a case study of the third provenance challenge is worthy of exploration. Such an application would provide an opportunity to compare our provenance collection mechanisms with other provenance systems in terms of being interoperable and answering proposed queries in the challenge.

8.4 Closing Note

This work is motivated by the need for studying provenance collection mechanisms of scientific workflow systems. Provenance is an increasingly important facility that enables scientific workflows to provide reproducibility, sharing and validation of experimental results. The design of a provenance collection mechanism was studied in this thesis, and a number of design dimensions identified and defined for it. We used the design dimensions to survey a set of existing provenance collection mechanisms. This work makes explicit the concept of a Model of Provenance; in particular, we proposed a multiple-granularity Model of Provenance, and demonstrated its use in our provenance collection mechanisms. This work addresses research problems concerned with adaptively collecting provenance information on workflows running over distributed environments and dealing with changes. This thesis has used the notion of separation of concerns, expressing adaptive-granularity and provenance-collection concerns in terms of both MOP and AOP software techniques. Both MOP and AOP oriented adaptive provenance collection mechanisms employ our multiple-granularity MoP which is compatible with the OPM MoP to enable interoperability of our mechanisms.

APPENDIX A: FINE-GRAINED PROVENANCE

In this section, the collected fine-grained provenance from our experiment is presented based on provided fine-grained MoP in section 4.5.2. Fine-grained provenance is a view of provenance account with account No. 3 because it is collected for fine-grained provenance. In the following, we just show one example of each OPM element, in addition to "I-WDF" dependency and intermediate artifact.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<opmGraph                                          xmlns="http://openprovenance.org/model/opmx#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns4="http://example.com/">
    <accounts>
        <account id="account3"/>
    </accounts>
    <processes>
        <process id="P_1">
            <account ref="account3"/>
            <label value="au.edu.adelaide.pna.processes.IntegerProducerWithToken@4c057cc6"/>
        </process>

</process>
    </processes>
    <artifacts>
        <artifact id="a_1">
            <account ref="account3"/>
            <label value="int"/>
            <annotation id="artifactAnnotation_1">
                <property key="tokenValue">
                    <value xsi:type="xsd:int">1</value>
                </property>
                <account ref="account2"/>
            </annotation>
        </artifact>

        <artifact id="IntermediateToken_1">
            <account ref="account3"/>
            <label value="int"/>
            <annotation id="IartifactAnnotation_1">
                <property key="tokenValue">
                    <value xsi:type="xsd:int">2</value>
                </property>
                <account ref="account3"/>
            </annotation>
        </artifact>

    </artifacts>
    <agents>
```

```
    <agent id="Agent_1">
        <account ref="account3"/>
        <label value="au.edu.adelaide.pna.system.ProcessThreadImpl@7b321ed6"/>
    </agent>

</agents>
<dependencies>
    <wasControlledBy id="WCB_1">
        <effect ref="P_1"/>
        <role value="DataFlow-Scheduler- scheduler"/>
        <cause ref="Agent_1"/>
        <account ref="account3"/>
    </wasControlledBy>

    <wasGeneratedBy id="WGB_1">
        <effect ref="a_1"/>
        <role value="out"/>
        <cause ref="P_1"/>
        <account ref="account3"/>
        <annotation id="wgbAnnotation_2">
            <property key="WGB - Port and Channel">
                <value xsi:type="EmbeddedAnnotation">
                    <value encoding="java.lang.Object@669c0287" id="channel">
                        <content xsi:type="xsd:string">Object</content>
                    </value>
                    <value encoding="0 =&gt; the only input port" id="port index">
                        <content xsi:type="xsd:int">0</content>
                    </value>
                </value>
            </property>
            <account ref="account3"/>
        </annotation>
    </wasGeneratedBy>

    <wasTriggeredBy id="WTB_1">
        <effect ref="P_3"/>
        <cause ref="P_2"/>
        <account ref="account3"/>
        <annotation id="wtbAnnotation_0">
            <property key="WTB - token coming into input port No. 1">
                <value xsi:type="xsd:string">1</value>
            </property>
            <account ref="account3"/>
        </annotation>
        <annotation id="wtbAnnotation_0">
            <property key="WTB - token coming into input port No. 2 ">
                <value xsi:type="xsd:string">1</value>
            </property>
            <account ref="account3"/>
```

```
            </annotation>
        </wasTriggeredBy>

<used id="USED_1">
            <effect ref="P_3"/>
            <role value="in1"/>
            <cause ref="a_1"/>
            <account ref="account3"/>
            <annotation id="usedAnnotation_1">
                <property key="USED - Port and Channel">
                    <value xsi:type="EmbeddedAnnotation">
                        <value id="channel"/>
                        <value encoding="1 =&amp;gt; there is two input ports" id="port index"/>
                    </value>
                </property>
                <account ref="account3"/>
            </annotation>
        </used>

        <wasDerivedFrom id="I-WDF_1">
            <effect ref="IntermediateToken_1"/>
            <cause ref="IntermediateToken_2"/>
            <account ref="account3"/>
            <annotation id="I-wdfAnnotation_1">
                <property key="I-WDB - process">
                                    <value
               xsi:type="xsd:string">au.edu.adelaide.pna.processes.ArithmaticProcessWithToken@
               534b58c</value>
                </property>
                <account ref="account3"/>
            </annotation>
        </wasDerivedFrom>

       <wasDerivedFrom id="I-WDF_1">
            <effect ref="IntermediateToken_1"/>
            <cause ref="IntermediateToken_2"/>
            <account ref="account3"/>
            <annotation id="I-wdfAnnotation_1">
                <property key="I-WDB - process">
                    <value
xsi:type="xsd:string">au.edu.adelaide.pna.processes.ArithmaticProcessWithToken@534b58c</value>
                </property>
                <account ref="account3"/>
            </annotation>
        </wasDerivedFrom>

    </dependencies>
    <annotations/>
</opmGraph>
```

APPENDIX B: MEDIUM-GRAINED PROVENANCE

The collected medium-grained provenance from our experiment is presented in this section. Medium-grained provenance is a view of provenance account with account No. 2 because it is collected for medium-grained provenance. In the following, we just show one example of each OPM element. The presented medium-grained MoP for collected medium-grained provenance is the same in both section 1.6.1 and 1.6.2.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<opmGraph                                          xmlns="http://openprovenance.org/model/opmx#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns4="http://example.com/">
    <accounts>
        <account id="account2"/>
    </accounts>
    <processes>
        <process id="P_1">
            <account ref="account2"/>
            <label value="au.edu.adelaide.pna.processes.IntegerProducerWithToken@173a30bd"/>
.....

</processes>
    <artifacts>
        <artifact id="a_2">
            <account ref="account2"/>
            <label value="int"/>
            <annotation id="artifactAnnotation_1">
                <property key="tokenValue">
                    <value xsi:type="xsd:int">1</value>
                </property>
                <account ref="account2"/>
            </annotation>
        </artifact>
.....

    </artifacts>
    <agents>
        <agent id="Agent_1">
            <account ref="account2"/>
            <label value="au.edu.adelaide.pna.system.ProcessThreadImpl@21832ae6"/>
        </agent>

.....

</agents>
```

```
    <dependencies>
        <wasControlledBy id="WCB_1">
            <effect ref="P_1"/>
            <role value="DataFlowStrategy- scheduler"/>
            <cause ref="Agent_1"/>
            <account ref="account2"/>
        </wasControlledBy>
.....

<wasGeneratedBy id="WGB_1">
            <effect ref="a_2"/>
            <role value="out"/>
            <cause ref="P_1"/>
            <account ref="account2"/>
            <annotation id="wgbAnnotation_1">
                <property key="WGB - Port and Channel">
                    <value xsi:type="EmbeddedAnnotation">
                        <value encoding="java.lang.Object@173775a1" id="channel">
                            <content xsi:type="xsd:string">Object</content>
                        </value>
                        <value encoding="0 =&gt; the only input port" id="port index">
                            <content xsi:type="xsd:int">0</content>
                        </value>
                    </value>
                </property>
                <account ref="account2"/>
            </annotation>
        </wasGeneratedBy>
.....


<wasTriggeredBy id="WTB_1">
            <effect ref="P_3"/>
            <cause ref="P_2"/>
            <account ref="account2"/>
            <annotation id="wtbAnnotation_0">
                <property key="WTB - token coming into input port No. 1">
                    <value xsi:type="xsd:string">1</value>
                </property>
                <account ref="account2"/>
            </annotation>
            <annotation id="wtbAnnotation_0">
                <property key="WTB - token coming into input port No. 2 ">
                    <value xsi:type="xsd:string">1</value>
                </property>
                <account ref="account2"/>
            </annotation>
        </wasTriggeredBy>
```

```
.....

<used id="USED_1">
            <effect ref="P_3"/>
            <role value="in1"/>
            <cause ref="a_2"/>
            <account ref="account2"/>
            <annotation id="usedAnnotation_1">
                <property key="USED - Port and Channel">
                    <value xsi:type="EmbeddedAnnotation">
                        <value id="channel"/>
                        <value encoding="1 =&gt; there is two input ports" id="port index"/>
                    </value>
                </property>
                <account ref="account2"/>
            </annotation>
        </used>
.....


<wasDerivedFrom id="WDF_1">
            <effect ref="a_7"/>
            <cause ref="a_2"/>
            <account ref="account2"/>
            <annotation id="wdfAnnotation_1">
                <property key="WDB - process">
                    <value
xsi:type="xsd:string">au.edu.adelaide.pna.processes.ArithmaticProcessWithToken@295cd6e5</value
>
                </property>
                <account ref="account2"/>
            </annotation>
        </wasDerivedFrom>


.....

    </dependencies>
    <annotations/>
</opmGraph>
```

APPENDIX C: COARSE-GRAINED PROVENANCE (SHORT VERSION)

In this section, the collected coarse-grained provenance from our experiment is presented based on provided coarse-grained MoP in section 4.5.2. Coarse-grained provenance is a view of provenance account with account No. 1 because it is collected for coarse-grained provenance. In the following, we just show one example of each OPM element.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<opmGraph xmlns="http://openprovenance.org/model/opmx#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns4="http://example.com/">
    <accounts>
        <account id="account1"/>
    </accounts>
    <processes>
        <process id="P_1">
            <account ref="account1"/>
            <label value="au.edu.adelaide.pna.processes.IntegerProducerWithToken@42d69a83"/>
.....


</processes>
    <artifacts>
        <artifact id="a_2">
            <account ref="account1"/>
            <label value="int"/>
        </artifact>
.....


</artifacts>
    <agents>
        <agent id="Agent_1">
            <account ref="account1"/>
            <label value="au.edu.adelaide.pna.system.ProcessThreadImpl@33802324"/>
        </agent>

.....

</agents>
    <dependencies>
        <wasControlledBy id="WCB_1">
            <effect ref="P_1"/>
```

```
            <role value="DataFlowStrategy- scheduler"/>
            <cause ref="Agent_1"/>
            <account ref="account1"/>
        </wasControlledBy>

.....

<wasDerivedFrom id="WDF_1">
            <effect ref="a_10"/>
            <cause ref="a_2"/>
            <account ref="account1"/>
        </wasDerivedFrom>

.....

    </dependencies>
    <annotations/>
</opmGraph>
```

APPENDIX D: COARSE-GRAINED PROVENANCE

In this section, the collected coarse-grained provenance from our experiment is presented based on provided coarse-grained MoP in section 4.5.1 is presented. Coarse-grained provenance is a view of provenance account with account No. 1 because it is collected for coarse-grained provenance. In the following, we just show one example of each OPM element.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<opmGraph                                          xmlns="http://openprovenance.org/model/opmx#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns4="http://example.com/">
    <accounts>
        <account id="account1"/>
    </accounts>
    <processes>
        <process id="P_1">
            <account ref="account1"/>
            <label value="au.edu.adelaide.pna.processes.IntegerProducerWithToken@42d69a83"/>
.....

</processes>
    <artifacts>
        <artifact id="a_2">
            <account ref="account1"/>
            <label value="int"/>
        </artifact>
.....

</artifacts>
    <agents>
        <agent id="Agent_1">
            <account ref="account1"/>
            <label value="au.edu.adelaide.pna.system.ProcessThreadImpl@33802324"/>
        </agent>
....

</agents>
    <dependencies>
        <wasControlledBy id="WCB_1">
            <effect ref="P_1"/>
            <role value="DataFlowStrategy- scheduler"/>
```

```
            <cause ref="Agent_1"/>
            <account ref="account1"/>
        </wasControlledBy>

.....

<wasGeneratedBy id="WGB_1">
            <effect ref="a_2"/>
            <role value="out"/>
            <cause ref="P_1"/>
            <account ref="account1"/>
        </wasGeneratedBy>

.....

     <wasTriggeredBy id="WTB_1">
            <effect ref="P_3"/>
            <cause ref="P_2"/>
            <account ref="account1"/>
        </wasTriggeredBy>

.....

<used id="USED_1">
            <effect ref="P_3"/>
            <role value="in1"/>
            <cause ref="a_2"/>
            <account ref="account1"/>
        </used>

.....

<wasDerivedFrom id="WDF_1">
            <effect ref="a_10"/>
            <cause ref="a_2"/>
            <account ref="account1"/>
        </wasDerivedFrom>

.....

    </dependencies>
    <annotations/>
</opmGraph>
```

APPENDIX E: MULTIPLE-GRANULARITY PROVENANCE

In this section, the collected multiple-granularity provenance from our experiment is presented based on provided multiple-granularity MoP in section 4.5.2 is presented.Multiple-granularity provenance is a view of provenance account with account No. 3, 2 and 1 because it contains for fine-, medium and coarse-grained provenance. In the following, we just show one example of each OPM element.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<opmGraph                                          xmlns="http://openprovenance.org/model/opmx#"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:ns4="http://example.com/">
    <accounts>
        <account id="account3"/>
        <account id="account2"/>
        <account id="account1"/>
    </accounts>
    <processes>
        <process id="P_1">
            <account ref="account3"/>
            <label value="au.edu.adelaide.pna.processes.IntegerProducerWithToken@73fb44da"/>
        </process>

…

</processes>
    <artifacts>
        <artifact id="a_1">
            <account ref="account3"/>
            <label value="515219417"/>
            <annotation id="artifactAnnotation_1">
                <property key="tokenValue">
                    <value xsi:type="xsd:int">1</value>
                </property>
                <account ref="account3"/>
            </annotation>
        </artifact>
<artifact id="a_17">
            <account ref="account2"/>
            <label value="1855889762"/>
            <annotation id="artifactAnnotation_17">
                <property key="tokenValue">
                    <value xsi:type="xsd:int">2</value>
                </property>
                <account ref="account2"/>
```

```
            </annotation>
        </artifact>

<artifact id="a_18">
            <account ref="account1"/>
            <label value="379415048"/>
        </artifact>
    </artifacts>
…
<agents>
        <agent id="Agent_1">
            <account ref="account3"/>
            <label value="au.edu.adelaide.pna.system.ProcessThreadImpl@5b19537a"/>
        </agent>

…

</agents>
    <dependencies>
        <wasControlledBy id="WCB_1">
            <effect ref="P_1"/>
            <role value="Dataflow- Scheduler"/>
            <cause ref="Agent_1"/>
            <account ref="account3"/>
        </wasControlledBy>

<wasGeneratedBy id="WGB_2">
            <effect ref="a_1"/>
            <role value="out"/>
            <cause ref="P_2"/>
            <account ref="account3"/>
            <annotation id="wgbAnnotation_1">
                <property key="WGB - Port and Channel">
                    <value xsi:type="EmbeddedAnnotation">
                        <value
encoding="au.edu.adelaide.pna.halfchannel.HalfChannelOutputPortImpl@70803a13" id="channel">
                            <content xsi:type="xsd:string">HalfChannelOutputPortImpl</content>
                        </value>
                        <value encoding="0 =&gt; the only input port" id="port index">
                            <content xsi:type="xsd:int">0</content>
                        </value>
                    </value>
                </property>
                <account ref="account3"/>
            </annotation>
        </wasGeneratedBy>

<wasTriggeredBy id="WTB_1">
            <effect ref="P_3"/>
```

```
            <cause ref="P_2"/>
            <account ref="account3"/>
            <annotation id="wtbAnnotation_0">
                <property key="WTB - token coming into input port No. 1">
                    <value xsi:type="xsd:string">1</value>
                </property>
                <account ref="account3"/>
            </annotation>
            <annotation id="wtbAnnotation_0">
                <property key="WTB - token coming into input port No. 2 ">
                    <value xsi:type="xsd:string">1</value>
                </property>
                <account ref="account3"/>
            </annotation>
        </wasTriggeredBy>


<used id="USED_1">
            <effect ref="P_3"/>
            <role value="in1"/>
            <cause ref="a_2"/>
            <account ref="account3"/>
            <annotation id="usedAnnotation_1">
                <property key="USED - Port and Channel">
                    <value xsi:type="EmbeddedAnnotation">
                        <value id="channel"/>
                        <value encoding="1 =&gt; there is two input ports" id="port index"/>
                    </value>
                </property>
                <account ref="account3"/>
            </annotation>
        </used>


<wasDerivedFrom id="WDF_1">
            <effect ref="a_9"/>
            <cause ref="a_2"/>
            <account ref="account3"/>
            <annotation id="wdfAnnotation_1">
                <property key="WDB - process">
                    <value
xsi:type="xsd:string">au.edu.adelaide.pna.processes.ArithmaticProcessWithToken@15c76b4c</value
>
                </property>
                <account ref="account3"/>
            </annotation>
        </wasDerivedFrom>
```

```
    <wasGeneratedBy id="WGB_13">
            <effect ref="a_9"/>
            <role value="out"/>
            <cause ref="P_4"/>
            <account ref="account2"/>
            <annotation id="wgbAnnotation_12">
                <property key="WGB - Port and Channel">
                    <value xsi:type="EmbeddedAnnotation">
                        <value
encoding="au.edu.adelaide.pna.halfchannel.HalfChannelOutputPortImpl@70297e0d" id="channel">
                            <content xsi:type="xsd:string">HalfChannelOutputPortImpl</content>
                        </value>
                        <value encoding="1 =&gt; there is two input ports" id="port index">
                            <content xsi:type="xsd:int">10</content>
                        </value>
                    </value>
                </property>
                <account ref="account2"/>
            </annotation>

<used id="USED_13">
            <effect ref="P_6"/>
            <role value="in1"/>
            <cause ref="a_9"/>
            <account ref="account2"/>
            <annotation id="usedAnnotation_13">
                <property key="USED - Port and Channel">
                    <value xsi:type="EmbeddedAnnotation">
                        <value id="channel"/>
                        <value encoding="1 =&gt; there is two input ports" id="port index"/>
                    </value>
                </property>
                <account ref="account2"/>
            </annotation>
        </used>


<wasDerivedFrom id="WDF_13">
            <effect ref="a_17"/>
            <cause ref="a_9"/>
            <account ref="account2"/>
            <annotation id="wdfAnnotation_13">
                <property key="WDB - process">
                    <value
xsi:type="xsd:string">au.edu.adelaide.pna.processes.ArithmaticProcessWithConstantWithToken@621
e3728</value>
                </property>
                <account ref="account2"/>
            </annotation>
```

```
        </wasDerivedFrom>


<wasTriggeredBy id="WTB_9">
            <effect ref="P_4"/>
            <cause ref="P_3"/>
            <account ref="account2"/>
            <annotation id="wtbAnnotation_0">
                <property key="WTB - token coming into input port ">
                    <value xsi:type="xsd:string">6</value>
                </property>
                <account ref="account2"/>
            </annotation>
        </wasTriggeredBy>


</wasTriggeredBy>
        <wasGeneratedBy id="WGB_30">
            <effect ref="a_16"/>
            <role value="out"/>
            <cause ref="P_6"/>
            <account ref="account1"/>
        </wasGeneratedBy>

<used id="U_18">
            <effect ref="P_7"/>
            <role value="in1"/>
            <cause ref="a_17"/>
            <account ref="account1"/>
        </used>

<wasDerivedFrom id="WDF_23">
            <effect ref="a_18"/>
            <cause ref="a_17"/>
            <account ref="account1"/>
        </wasDerivedFrom>

<wasGeneratedBy id="WGB_31">
            <effect ref="a_20"/>
            <role value="out"/>
            <cause ref="P_5"/>
            <account ref="account1"/>
        </wasGeneratedBy>


</dependencies>
    <annotations/>
</opmGraph>
```

APPENDIX F: KEPLER PROVENANCE RECORDER CONFIGURATION

There are four implementations for “Recording” interface, firstly, the “TextFileRecording” (named “Text File” in configuration file) implements methods of “Recording” interface to write the provenance data into a text file. “SQLRecording” (named “SQL-SPA” in configuration file), “SQLRecordingV7” (named “SQL-SPA-v7” in configuration file), “SQLRecordingV8” (named “SQL-SPA-v8” in configuration file) are the other three implementations of “Recording” that record data into database. The “SQLRecordingV7” has an advantage of tracking the changes in the workflow structure in comparison with “SQLRecording”, while “SQLRecordingV8” add more features to “SQLRecordingV7” version. These features include

- Saving the workflow contents for each execution.
- Saving contents of files referenced in tokens.
- Each workflow and workflow execution has unique LSID.
- Workflow name no longer required.
- Explicit mapping from token write to token read.
- Combining actor-firing events into a single "firing cycle event".
- Recording execution exceptions.
- Storing associated data for executions.

The “SQLQueryV8” establishes a connection to database and prepares a number of queries. This class is appropriate place for writing customized queries.

The “OpenProvenanceModelXML” (named “OPM XML” in configuration file) records based on Open Provenance Model (OPM) XML. It records execution start and stop, actor firing and port events; register ports and actors; add Parameters for “ProvenanceListener”; and react to parameter changes.

There is two ways to enable and configure Provenance Recorder actor into Kepler. First and most convenient way is by drag and drops the provenance actor into Kepler GUI design canvas like Figure F.1. Then opens this actor and changes the configurations include, workflow name, database configuration and recording type (here, “SQLRecordingV8” is chosen by selecting SQL-SPA-V8). The second way to enable provenance recorder is creating

an object of ProvenanceRecorder class in constructor of an actor's source code. In this way, the ProvenanceRecorder get the default configuration of confiquration.xml file then establish a connection to database base on provided information and store the recorded provenance information into database tables.

The configuration of provenance recorder can be changed in the configuration.xml file in resource folder of provenance package. In this way, the changed values are default value that can be helpful for easily reusing this configuration. Moreover, this model can be more useful when provenance actor is enabled from source code. In this case, just by create an object of the provenance recorder class, all default configuration are usable in provenance recorder.

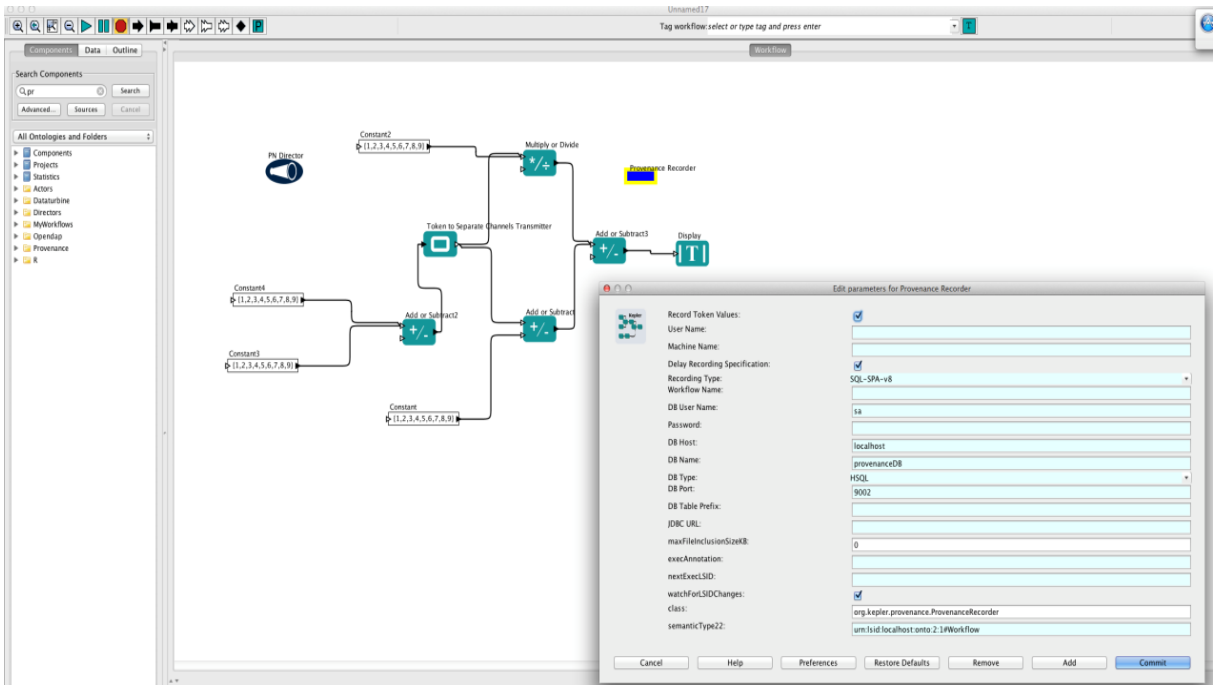


Figure F.1. Configuration GUI of Provenance Recorder.

APPENDIX G: A SIMPLE CASE STUDY USING ENIGMA

In this section, we will explore the mechanism of Dynamic proxy in Enigma and then a simple case study of Enigma will be presented. The purpose of this section is to provide detailed information on how Enigma operates based on dynamic proxy. We present a detailed (trace of) reification process in Enigma. This section is quite detailed that is referred by section 6.3.2.1 and 7.2.

Java Dynamic Proxy is used in Enigma as a reification mechanism in order to create object dynamically. It enables Enigma to transparently intercept and redirect method invocations to meta-level objects. Enigma uses Dynamic proxy class to implement a list of interfaces, which are specified at runtime when classes are created. Dynamic proxy contains a couple of elements, which are defined behaviours and working mechanism,

1. Proxy instance: is a proxy class instance.
2. Proxy interface: is an interface that is implemented by a proxy class.
3. Invocation handler: is an object associated with each proxy instance and implements the interface “InvocationHandler”. When a method invocation occurs on a proxy instance through one of the proxy interfaces, it is dispatched to “invoke” method of the instance's invocation handler. Then, eventually, passing to Java reflection method object to identify the invoked method and return the array of objects and arguments.

“DynamicMetaFactory” class equips Enigma meta-object protocol with using Dynamic proxy. “DynamicMetaFactory” class creates a meta-level object and a proxy object. The proxy object is used in place of the normal object and redirects invoked methods on the object. “DynamicMetaFactory” class implements “newMetaLevelFor” method to create meta-objects (Figure G.1). This class intercepts interfaces that are implemented by base-object via using Java Reflection API. MethodReifier, metaObject and metalevelInterface interfaces are also added to DynamicMetaFactory’s interface set. “newProxyInstance” method plays the proxy instance role. It takes interface set and base-object (that implements Dynamic Proxy API’s InvocationHandler interface) as input parameters. Then it returns a proxy object that implements interfaces and delegates all calls to “InvocationHandler”.

```
public Object newMetaLevelFor(Object baseObject,MethodReifier reifier)
{
        Set interfaceSet = new HashSet();
        Toolkit.getAllInheritedInterfaces(interfaceSet,baseObject.getClass());
        interfaceSet.add(MethodReifier.class);
        interfaceSet.add(MetaObject.class);
        interfaceSet.add(MetaLevelInterface.class);
        Class[] interfaces = Toolkit.setToClassArray(interfaceSet);
        Object baseObjectProxy =
        Proxy.newProxyInstance(baseObject.getClass().getClassLoader(),interfaces,reifier);
        return baseObjectProxy;
}
```

Figure G.1. newMetaLevelFor method in DynamicMetaFactory class.

"StartPointMetaObject" is an InvocationHandler implementation in Enigma. The "StartPointMetaObject" accomplishes the meta-object method invocation reifier duties in Enigma architecture. As a reifier, it converts the intercepted method invocation (by proxy) into a "MetaMethodInvocation", following that passes it to composition of "MetaMarshalHandler" meta-object.

In the next step, "StartPointMetaObject" drives the invocation through a composition of "MetaTransmitHandler" metaobject to an "EndPointMetaObject", which plays the destination base-objects' meta-object. The "EndPointMetaObject" is determined, when the "MetaMethodInvocation" invokes method upon destination base-object or meta-object, or either invokes the reification upon itself. In a case that, the invocation occurs upon destination baseobject, it causes reification upon the composition of "MetaExecutionHandler" meta-object. The "EndPointMetaObject" returns result, in terms of "MetaValueResult" or "MetaExecutionResult", back through the meta-level to the StartPointMetaObject, which converts the result to a value and delivers to the base-object.

Case study

In this case study, as shown in Figure G.2, an integer parameter is created and initiated with an integer value (=5), and then the parameter is shown in output console. In the following, the creation mechanism of meta-object and meta-level in Enigma for this case study are presented. The case study has a race Execution meta-behaviour.

Figure G.5 presents more detailed information on meta-level and meta-object creation. According to the first two lines of source code in Figure G.3, which are shown in steps 12 and

14 in the Figure G.4, meta-object and methodreifier interfaces are implemented at runtime with the help of proxy object. Following that in second line of Figure G.2, third line in Figure 14 and step 16 in Figure G.4, Trace Execution meta-behaviour is added to the reflection model. "TraceExecutionHandlerClass" in Figure G.3 creates an object for "TraceExecutionHandler" class, which it eventually causes to invoke "updateMetaInformation" method on meta-object ("metaObject.updateMetaInformation"). This invocation is intercepted by proxy and delegated to same method on proxy ("proxy.updateMetaInformation") (steps 16 to 35 in Figure G.4). The same scenario happens for the last line of Figure G.3 for invoking "addMetaExecutionHandler" on metaobject, as well (steps 36 to 54 in Figure G.4).

```
public class CustomizeDebugTest {

    public static void main(String args[]) {
        MetaFactory factory = new DynamicMetaFactory();
        factory.addMetaLevelCustomization(Object.class, new TraceExecutionHandlerClass());
        Foo foo = (Foo) factory.newMetaLevelFor(new FooImpl());
        foo.add(new Integer(5));
        System.out.println("List element 0=" + foo.get(0));
    }
}
```

Figure G.2. Main class of the case study.

```
public class TraceExecutionHandlerClass implements MetaHandlerClass
{
        public void customizeMetaObject(MetaLevelInterface metalevelInterface)
        {
                MetaObject metaObject = metalevelInterface.getTargetMetaObject();
                MethodReifier mr=metalevelInterface.getThisMethodReifier();
                TraceExecutionHandler handler = new TraceExecutionHandler(metaObject);
                metaObject.addMetaExecutionHandler(handler);
        }
```

Figure G.3. Trace meta-behavour.

All methods calls in the reflection model are delegated to "StartPointMetaObject", which it implements "MethodReifier" interface. While it delegated to "SerializableMethodInvocation" class which uses basic Java reflection method ("java.lang.reflect.Method") to wrap methods with a couple of metainformation about method. In a case that, the delegation to "SerializableMethodInvocation" class reaches to "Invoke" method of this class. The "invoke" method behaves differently in different phases. It returns "metaResult" during execution phase, while during marshalling and transmission phases cause changing the phase from marshalling to transmission and then from transmission to execution (steps 22 and 41 in Figure G.4).

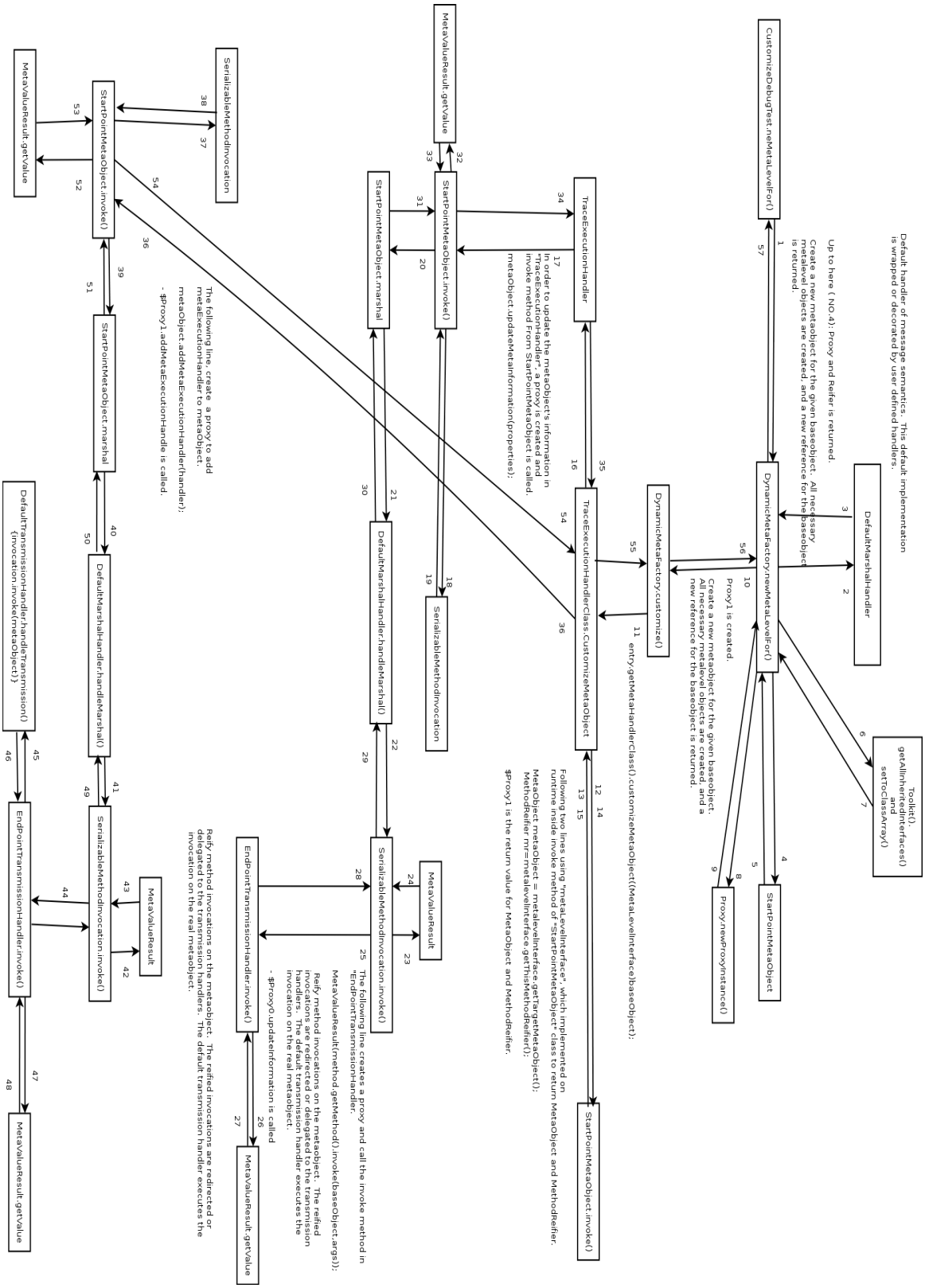


Figure G.4. Creation of meta-object and meta-level.

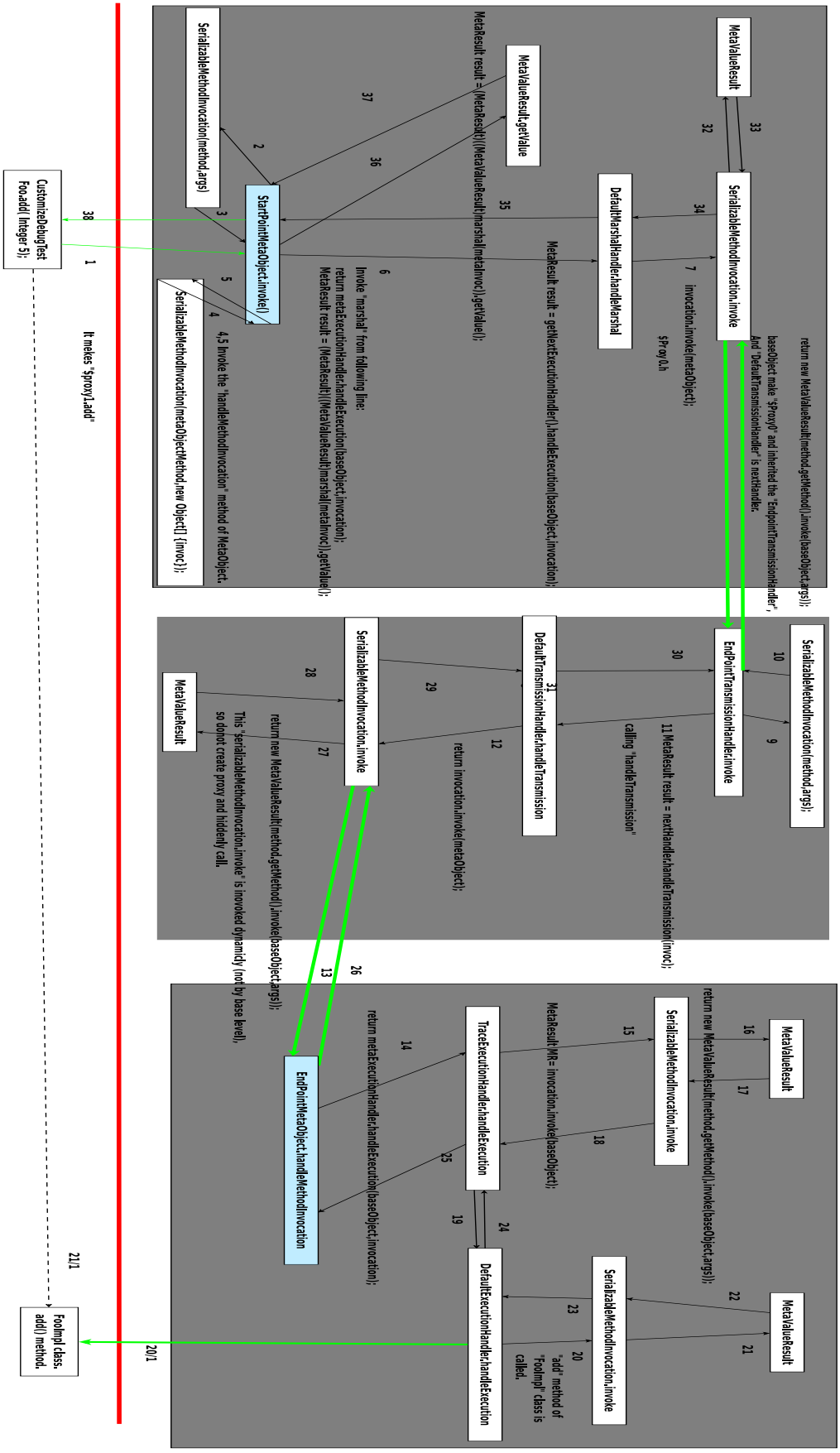


Figure G.5. Message decomposition in Enigma.

Method invocation through Enigma MOP

Method invocation like “Foo.add(Integer 5)” method in “CustomizeDebugTest” class Figure G.2, is demonstrated through the Enigma meta-object protocol. According to abovementioned information (Figure G.2, G.3 and G.4) “newMetaLevelFor” method, which is implemented in “DynamicMetafactory” class, creates a meta-level with appropriate meta-objects and the Trace Execution meta-behaviour is added to the reflection model. At this stage, model provides meta-level and meta-object for method invocation. The method invocation is processed in three marshal, transmission and execution phases (Figure G.5).

In the marshalling phase, as add method from “FooImpl” object is called, a proxy for this method is created. The Invoke method of “StartPointMetaObject” class plays the invocation handler role and intercepts the method invocation on a proxy for the base-object and delegates the reified invocation to handle marshal in “DefaultMarshalHandler” class. The invoke method in “StartPointMetaObject” class creates a HandleMethodInvocation object by “serializableMethodInvocation”, which is a Java wrapper class to add some basic information about method in step 4 of Figure G.6.

In the following step (step 6), the handleMarshal method of Default---handler (here is “defaultMarshalHandler”) class calls the “invoke” method of “serializableMethodInvocation” to return the “MetaResult” (the object of serializableMethodInvocation class will not specify until invocation time). This “invoke” method is reified by basic Java reflection method and makes a new proxy for that. In the step 8, the proxy delegates the “invoke” method of next handler class (here, handler class is “EndPointTransmissionHandler”). “EndPointTransmissionHandler” implements “MetaLevelObject” and “InvocationHandler” interfaces. In this step, the method invocation passes into transmission phase from marshal phase.

In the transmission phase, similar scenario like marshalling phase happens. In this phase, invoked methods of handler class wants to return the MetaResult which is in Dafault***handler (here is DefaultTranssmissionHandler). Similar to other handler classes, DefaultTranssmissionHandler contains “invoke” method that is delegated to “invoke” method in SerializableMethodInvocation. This “invoke” method changes the phase. In this stage, it

turns into execution phase and calls the “handleMethodInvocation” method of “EndPointMetaObject”.
In the execution phase, the “EndPointMetaObject” implements “MetaObject” to act as a metaobject of destination baseobject. Moreover, Metaobject reflects metainformation and metabehaviour of an object on the baselevel. The “handleMethodInvocation” of this class calls “handleExecution” method of “TraceExecutionHandler” class, in order to print some debug message before and after invocation. This “handleExecution” method is delegated to same method of “DefaultExecutionHandler” class in step 19. The “DefaultExecutionHandler” calls the “invoke” method of “SerializableMethodInvocation“(like other Default—handler classes).

In step 20 and 21, the “invoke” method returns actual “metaResult” because it is the time of execution. While previously this invoke methods is just called and waiting for invocation time to specify the target object and return results. In this time, add method of “FooImpl” class is called on destination base-object and meta-object. As a consequence, methods in method call stack (Figure G.2), which was waiting for specifying target object could be run and particularly the “invoke” method of “SerializableMethodInvocation” could return the “metaResult” and complete processes up to the first method invocation at “customizedDebudTest“ class (step 22 to 38).

There is not any direct return link to caller base-object in Figure G.5, because this method invocation (“Foo.add”) does not return any values. If it does need to return any value (like “Foo.get()” method), it would be necessary to have a direct return link between “FooImpl” and “customizedDebudTest” classes. However, the reified results either “MetaValueResult” or “MetaExecutionResult” are returned back though the metalevel in any situation.

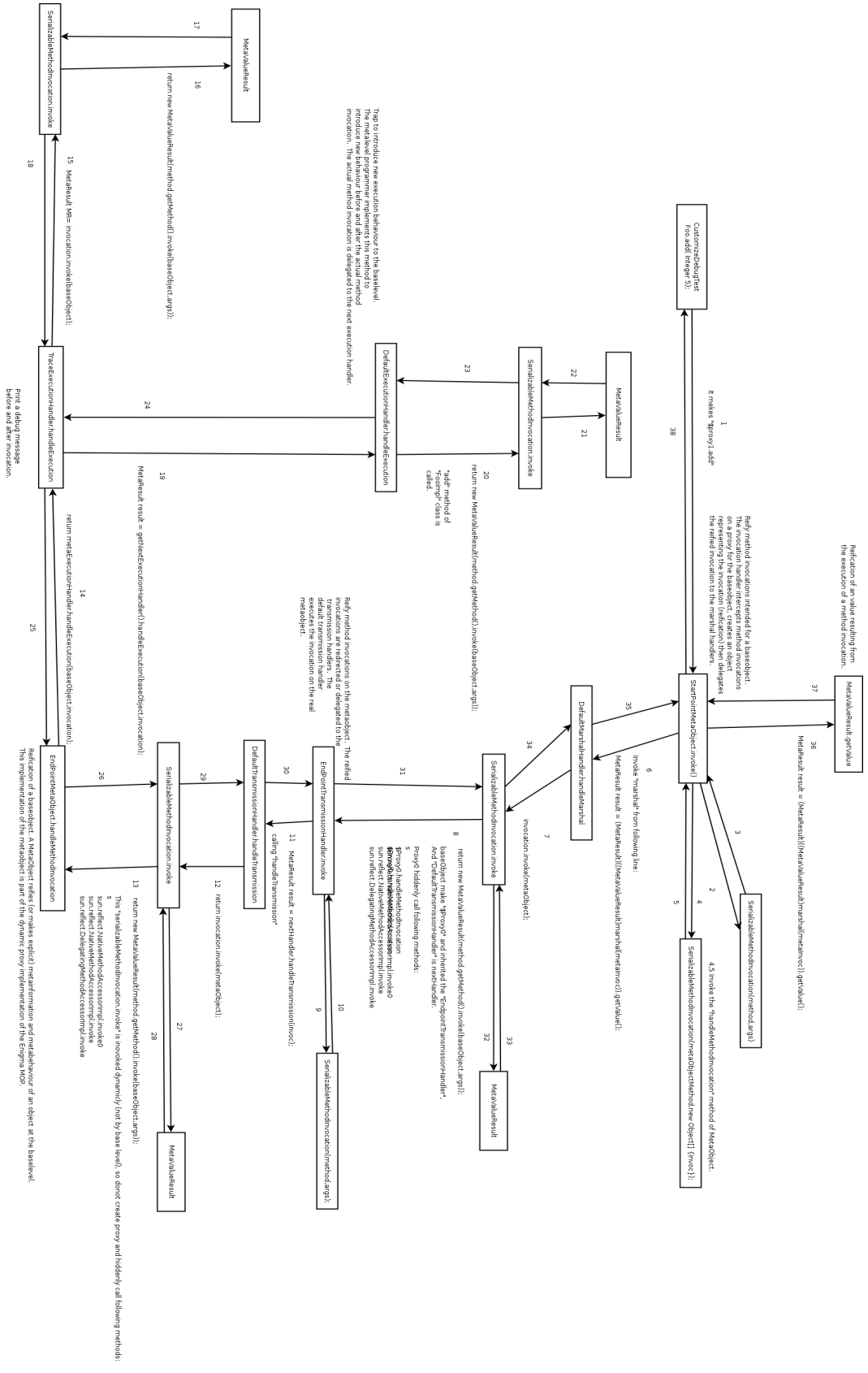


Figure G.6. Trace of method invocation through Enigma message decomposition.

REFERENCES

[1] A. J. G. Hey, S. Tansley, and K. M. Tolle, *The fourth paradigm: data-intensive scientific discovery*, 1st ed. Washington: Microsoft Research Redmond, 2009.

[2] A. Barker and J. van Hemert, "Scientific Workflow: A Survey and Research Directions," in Parallel Processing and Applied Mathematics. vol. 4967, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, Eds., Berlin, Germany: Springer, 2008, pp. 746-753.

[3] B. Ludascher, I. Altintas, S. Bowers, J. Cummings, T. Critchlow, E. Deelman*, et al.*, "Scientific process automation and workflow management," *Scientific Data Management: Challenges, Existing Technology, and Deployment, Computational Science Series, chapter 13. Chapman and Hall/CRC,* 2009.

[4] Y. Simmhan, R. Barga, C. Van Ingen, E. Lazowska, and A. Szalay, "Building the trident scientific workflow workbench for data management in the cloud," in Proceeding of *3rd International Conference on Advanced Engineering Computing and Applications in Sciences (ADVCOMP)*, 2009, pp. 41-50.

[5] Y. Simmhan, C. Van Ingen, G. Subramanian, and J. Li, "Bridging the gap between desktop and the cloud for escience applications," in Proceeding of *IEEE 3rd International Conference on Cloud Computing (CLOUD)*, 2010, pp. 474-481.

[6] M. Shields, "Control- Versus Data-Driven Workflows," in Workflows for e-Science, I. Taylor, E. Deelman, D. Gannon, and M. Shields, Eds., London: Springer, 2007, pp. 167-173.

[7] C. Lin, S. Lu, X. Fei, A. Chebotko, D. Pai, Z. Lai*, et al.*, "A reference architecture for Scientific workflow management systems and the VIEW SOA solution," *IEEE Transactions on Services Computing,* vol. 2, pp. 79-92, 2009.

[8] F. Ranno and S. Shrivastava, "A review of distributed workflow management systems," in Proceeding of *the international joint conference on Work activities coordination and collaboration (WACC99)*, San Francisco, California, 1999.

[9] Q. Wu, M. Zhu, Y. Gu, P. Brown, X. Lu, W. Lin*, et al.*, "A distributed workflow management system with case study of real-life scientific applications on Grids," *Journal of Grid Computing,* vol. 10, pp. 367-393, 2012.

[10] J. A. Miller, A. P. Sheth, K. J. Kochut, and X. Wang, "CORBA-based run-time architectures for workflow management systems," *Journal of Database Management (JDM),* vol. 7, pp. 16-27, 1996.

[11] H. Li, Y. Yang, and M. Shi, "Key Issues and Experiences in Development of Distributed Workflow Management Systems," in Web Technologies and Applications. vol. 2642, X. Zhou, M. Orlowska, and Y. Zhang, Eds., Berlin, Germany: Springer, 2003, pp. 507-512.

[12] K. Görlach, M. Sonntag, D. Karastoyanova, F. Leymann, and M. Reiter, "Conventional Workflow Technology for Scientific Simulation," in Guide to e-Science, X. Yang, L. Wang, and W. Jie, Eds., London: Springer, 2011, pp. 323-352.

[13] C. Hahn, S. Horn, S. Jablonski, R. Lay, J. Neeb, R. Schamburger*, et al.*, "Taxonomy of distribution concepts for Workflow Management," University Erlangen-Nürnberg, Technical Report: P39.

[14] I. Altintas, J. Wang, D. Crawl, and W. Li, "Challenges and approaches for distributed workflow-driven analysis of large-scale biological data," in Proceeding of *Joint EDBT/ICDT Workshops*, Berlin, Germany, 2012, pp. 73-78.

[15] H. Zheng and X. Yin, "The Architecture Design of a Distributed Workflow System," in Proceeding of *11th International Symposium on Distributed Computing and Applications to Business, Engineering & Science (DCABES)*, Guilin, China, 2012, pp. 9-12.

[16] R. Barga, Y. Simmhan, E. C. Withana, S. Sahoo, J. Jackson, and N. Araujo, "Provenance for Scientific Workflows Towards Reproducible Research," *IEEE Data Engineering Bulleting,* vol. 33, pp. 50-59, 2010.

[17] S. Islam, "Provenance, Lineage, and Workflows," M.S. thesis, Department of Computer Science Department, Brown University, Providence, Rhode Island, 2010.

[18] J. Kim, E. Deelman, Y. Gil, G. Mehta, and V. Ratnakar, "Provenance trails in the wings/pegasus system," *Concurrency and Computation: Practice and Experience,* vol. 20, pp. 587-597, 2008.

[19] S. M. S. da Cruz, P. M. Barros, P. M. Bisch, M. L. M. Campos, and M. Mattoso, "Provenance services for distributed workflows," in Proceeding of *8th IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, Lyon, France, 2008, pp. 526-533.

[20] S. B. Davidson, S. C. Boulakia, A. Eyal, B. Ludäscher, T. M. McPhillips, S. Bowers*, et al.*, "Provenance in Scientific Workflow Systems," *IEEE Data Engineering Bulletin,* vol. 30, pp. 44-50, 2007.

[21] J. Freire, D. Koop, E. Santos, and C. T. Silva, "Provenance for computational tasks: A survey," *Computing in Science & Engineering,* vol. 10, pp. 11-21, 2008.

[22] M. Ooms, "Provenance management in practice," M.S. thesis, Department of Electrical Engineering, Mathematics and Computer Science, University of Twente, Enschede, Netherlands, 2009.

[23] W. C. Tan, "Research Problems in Data Provenance," *IEEE Data Engineering Bulletin,* vol. 27, pp. 45-52, 2004.

[24] S. Bowers, "Scientific workflow, provenance, and data modeling challenges and approaches," *Journal on Data Semantics,* vol. 1, pp. 19-30, 2012.

[25] J. Zhao, F. Sun, C. Torniai, A. Bakshi, and V. Prasanna, "A Provenance-Integration Framework for Distributed Workflows in Grid Environments," in Proceeding of *Workshop on Grid and Utility Computing*, Bangalore, India, 2008, pp. 17-20.

[26] S. B. Davidson and J. Freire, "Provenance and scientific workflows: challenges and opportunities," in Proceeding of *ACM SIGMOD international conference on Management of data*, Vancouver, Canada, 2008, pp. 1345-1350.

[27] J. Dennis, "Data Flow Computation," in Control Flow and Data Flow: Concepts of Distributed Programming. vol. 14, M. Broy, Ed., Berlin, Germany: Springer, 1986, pp. 345-398.

[28] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in Proceeding of *16th International Conference on Scientific and Statistical Database Management*, 2004, pp. 423-424.

[29] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous concurrent modeling and design in java (volume 3: Ptolemy ii domains)," EECS Department, University of California, Berkley, California, Technical Report: UCB/EECS-2008-37, 2008.

[30] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng*, et al.*, "Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii)," EECS Department, University of California, Berkley, California, Technical Report: UCB/EECS-2008-28, 2008.

[31] J. Yu, R. Buyya, and K. Ramamohanarao, "Workflow scheduling algorithms for grid computing," in Metaheuristics for scheduling in distributed computing environments. vol. 146, F. Xhafa and A. Abraham, Eds., Berlin, Germany: Springer, 2008, pp. 173-214.

[32] A. Avanes and J.-C. Freytag, "Adaptive workflow scheduling under resource allocation constraints and network dynamics," *Very Large Data Base (VLDB) Endowment,* vol. 1, pp. 1631-1637, 2008.

[33] P. Senkul and I. H. Toroslu, "An architecture for workflow scheduling under resource allocation constraints," *Information Systems,* vol. 30, pp. 399-422, 2005.

[34] R. Buyya, D. Abramson, J. Giddy, and H. Stockinger, "Economic models for resource management and scheduling in Grid computing," *Concurrency and Computation: Practice and Experience,* vol. 14, pp. 1507-1542, 2002.

[35] J. Delaney, G. Heath, A. Chave, B. Howe, and H. Kirkham, "NEPTUNE: Real-time ocean and earth sciences at the scale of a tectonic plate," *Oceanography,* vol. 13, pp. 71-79, 2000.

[36] W. I. Grosky, A. Kansal, S. Nath, L. Jie, and Z. Feng, "SenseWeb: An Infrastructure for Shared Sensing," *IEEE MultiMedia,* vol. 14, pp. 8-13, 2007.

[37] J. Frew, D. Metzger, and P. Slaughter, "Automatic capture and reconstruction of computational provenance," *Concurrency and Computation: Practice and Experience,* vol. 20, pp. 485-496, 2008.

[38] Web-Page. (2015, 1 March). *ES3 Project*. Available: http://people.eri.ucsb.edu/~es3/public/new/htdocs/projects/proj-essw.htm

[39] D. N. Williams, T. Bremer, C. Doutriaux, J. Patchett, S. Williams, G. Shipman*, et al.*, "Ultrascale Visualization of Climate Data," *Computer,* vol. 46, pp. 68-76, 2013.

[40] Web-Page. (2015, 20 March). *vistrails*. Available: http://www.vistrails.org/index.php/Main_Page

[41] C. T. Silva, J. Freire, and S. P. Callahan, "Provenance for visualizations: Reproducibility and beyond," *Computing in Science & Engineering,* vol. 9, pp. 82-89, 2007.

[42] C. Scheidegger, D. Koop, E. Santos, H. Vo, S. Callahan, J. Freire*, et al.*, "Tackling the provenance challenge one layer at a time," *Concurrency and Computation: Practice and Experience,* vol. 20, pp. 473-483, 2008.

[43] Web-Page. (2014). *Provenance Browse*. Available: http://www.vistrails.org/usersguide/v2.1/html/provenance.html - fig-browser

[44] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance in e-science," *ACM Sigmod Record,* vol. 34, pp. 31-36, 2005.

[45] Y. L. Simmhan, B. Plale, and D. Gannon, "A survey of data provenance techniques," Computer Science Department, Indiana University, Bloomington, Indiana, Technical Report: IUB-CS-TR618, 2005.

[46] I. Altintas, O. Barney, and E. Jaeger-Frank, "Provenance Collection Support in the Kepler Scientific Workflow System," in Provenance and Annotation of Data. vol. 4145, L. Moreau and I. Foster, Eds., Berlin, Germany: Springer, 2006, pp. 118-132.

[47] S. M. S. da Cruz, M. L. M. Campos, and M. Mattoso, "Towards a taxonomy of provenance in scientific workflow management systems," in Proceeding of *IEEE Congress on Services*, Los Angeles, California, 2009, pp. 259-266.

[48] B. Glavic and K. R. Dittrich, "Data Provenance: A Categorization of Existing Approaches," in Proceeding of *Conference on Datenbanksysteme in Buisness, Technologie und Web (BTW)*, Aachen, Germany, 2007, pp. 227-241.

[49] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, *et al.*, "The Open Provenance Model core specification (v1.1)," *Future Generation Computer Systems,* vol. 27, pp. 743-756, Jun 2011.

[50] P. Groth, S. Miles, P. Missier, and L. Moreau. (2009, June 04). *A proposal for handling collections in the open provenance model*. Available: http://mailman.ecs.soton.ac.uk/pipermail/provenance-challenge-ipaw-info/attachments/20090605/85b3e182/attachment-0001.pdf

[51] P. Groth and L. Moreau. (2013, April 30). *PROV-Overview*. Available: http://www.w3.org/TR/prov-overview/

[52] D. Garijo and Y. Gil. (2012, September 10). *The OPMW Ontology*. Available: http://www.opmw.org/model/OPMW_20121009/

[53] Y. Simmhan, P. Groth, and L. Moreau, "Special section: The third provenance challenge on using the open provenance model for interoperability," *Future Generation Computer Systems,* vol. 27, pp. 737-742, 2011.

[54] D. Crawl, J. Wang, and I. Altintas, "Provenance for MapReduce-based data-intensive workflows," in Proceeding of *6th workshop on Workflows in support of large-scale science*, Seattle, Washington, 2011, pp. 21-30.

[55] S. M. S. d. Cruz, C. E. Paulino, D. d. Oliveira, M. L. M. Campos, and M. Mattoso, "Capturing distributed provenance metadata from cloud-based scientific workflows," *Journal of Information and Data Management,* vol. 2, pp. 43-50, 2011.

[56] K.-K. Muniswamy-Reddy, P. Macko, and M. I. Seltzer, "Provenance for the Cloud," in Proceeding of *the 8th USENIX conference on File and storage technologies*, San Jose, California, 2010, pp. 15-14.

[57] K.-K. Muniswamy-Reddy, P. Macko, and M. I. Seltzer, "Making a Cloud Provenance-Aware," in Proceeding of *Workshop on the Theory and Practice of Provenance*, San Francisco, California, 2009.

[58] A. Marinho, L. Murta, C. Werner, V. Braganholo, S. M. S. d. Cruz, E. Ogasawara, *et al.*, "ProvManager: a provenance management system for scientific workflows," *Concurrency and Computation: Practice and Experience,* vol. 24, pp. 1513-1530, 2012.

[59] C. V. Lopes, "Aspect-Oriented Programming: An historical perspective (what's in a name?)," University of California, Irvine, California, Technical Report: UCI-ISR-02-5, 2002.

[60] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng, "Composing adaptive software," *Computer,* vol. 37, pp. 56-64, 2004.

[61] I. R. Forman and N. Forman, *Java reflection in action*: Manning Publications Co., 2004.

[62] P. Maes, "Concepts and experiments in computational reflection," in Proceeding of *Object-oriented programming systems, languages and applications (OOPSLA)*, Orlando, Florida, USA, 1987, pp. 147-155.

[63] A. Oliva, I. C. Garcia, and L. E. Buzato, "The Reflective Architecture of Guaraná," State University of Campinas, São Paulo, Brazil, Technical Report: IC-98-14, 1998.

[64] A. Corradi, E. Lodolo, S. Monti, and S. Pasini, "Dynamic reconfiguration of middleware for ubiquitous computing," in Proceeding of *the 3rd international workshop on Adaptive and dependable mobile ubiquitous systems*, London, United Kingdom, 2009, pp. 7-12.

[65] B. C. Smith, "Reflection and semantics in Lisp," in Proceeding of *the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL84)*, Salt Lake City, Utah, USA, 1984, pp. 23-35.

[66] G. Coulson, "What is reflective middleware," *IEEE Distributed Systems Online,* vol. 2, pp. 165-169, 2001.
[67] R. Barbosa and L. M. Pinho, "Monitoring of Real time Systems: a case for Reflection," School of Engineering, Polytechnic Institute of Porto, Porto, Portugal, Technical Report: HURRAY-TR-0413, 2004.
[68] M. Aksit and Z. Choukair, "Dynamic, adaptive and reconfigurable systems overview and prospective vision," in Proceeding of *the 23rd International Conference on Distributed Computing Systems Workshops*, 2003, pp. 84-89.
[69] R. Pawlak, L. Seinturier, J.-P. Retaillé, and H. Younessi, *Foundations of AOP for J2EE Development*: Apress, 2005.
[70] T. Elrad, M. Aksit, G. Kiczales, K. J. Lieberherr, and H. Ossher, "Discussing aspects of AOP," *Communications of the ACM,* vol. 44, pp. 33-38, 2001.
[71] P. T. Groth, "The Origin of Data," Ph.D. dissertation, Department of Electronics and Computer Science, University of Southampton, Southampton, England, 2007.
[72] K.-K. Muniswamy-Reddy, "Foundations for provenance-aware systems," Ph.D. dissertation, Department of Engineering and Applied Sciences, Harvard University Cambridge, Massachusetts, 2010.
[73] D. A. Koop, "Managing Provenance For Knowledge Discovery and reuse," Ph.D. dissertation, Department of Computer Science, The University of Utah, Salt Lake City, Utah, 2012.
[74] Web-Page. (2009, April 09). *Provenance Challenge 3 Questions*. Available: http://twiki.ipaw.info/bin/view/Challenge/ProvenanceQuestionsPc3
[75] L. Moreau, N. Kwasnikowska, and J. Van den Bussche. (2009). *The foundations of the open provenance model*. Available: http://eprints.soton.ac.uk/id/eprint/267282
[76] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen, "Putting lipstick on pig: enabling database-style workflow provenance," *Very Large Data Base (VLDB) Endowment,* vol. 5, pp. 346-357, 2011.
[77] M. Stamatogiannakis, P. Groth, and H. Bos, "Looking Inside the Black-Box: Capturing Data Provenance using Dynamic Instrumentation," in Provenance and Annotation of Data and Processes. vol. 8628, B. Ludäscher and B. Plale, Eds., Switzerland: Springer International Publishing, 2015, pp. 155-167.
[78] P. T. Groth, "On the Record: Provenance in Large Scale, Open Distributed Systems," A mini-thesis for transfer from MPhil to PhD, Department of Electronics and Computer Science, University of Southampton, Southampton, England, 2005.
[79] Web-Page. (2015, March 10). *Concurrent versions system*. Available: http://www.cvshome.org/
[80] C. M. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick, *Version control with subversion*, 2nd ed.: O'Reilly Media, Inc., 2008.
[81] B. Cornell, P. A. Dinda, and F. E. Bustamante, "Wayback: A user-level versioning file system for linux," in Proceeding of *USENIX Annual Technical Conference, FREENIX Track*, Boston, Massachusetts, 2004, pp. 19–28.
[82] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C. Veitch, R. W. Carton, and J. Ofir, "Deciding when to forget in the Elephant file system," *SIGOPS Operating Systems Review,* vol. 33, pp. 110-123, 1999.
[83] D. P. Lanter, "Lineage in gis: The problem and a solution," National Center for Geographic Information and Analysis (NCGIA), Santa Barbara, California, Technical Report: TR90-6, 1990.
[84] A. P. Marathe, "Tracing lineage of array data," *Journal of Intelligent Information Systems,* vol. 17, pp. 193-214, 2001.

[85] A. Vahdat and T. E. Anderson, "Transparent result caching," in Proceeding of *USENIX Annual Technical Conference*, New Orleans, Louisiana, 1998.
[86] J. Widom, "Trio: A system for integrated management of data, accuracy, and lineage," in Proceeding of *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California, 2005.
[87] H. Fan, "Tracing data lineage using schema transformation pathways," in Advances in Databases. vol. 2405, B. Eaglestone, S. North, and A. Poulovassilis, Eds., Berlin, Germany: Springer 2003, pp. 50-53.
[88] R. Ikeda and J. Widom, "Panda: A System for Provenance and Data," *IEEE Data Engineering Bulletin,* vol. 33, pp. 42-49, 2010.
[89] D. Kranzlmuller, "Dewiz-event-based debugging on the grid," in Proceeding of *10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, 2002, pp. 162-169.
[90] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer, "Dynamic monitoring of high-performance distributed applications," in Proceeding of *11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11)*, 2002, pp. 163-170.
[91] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: design, implementation, and experience," *Parallel Computing,* vol. 30, pp. 817-840, 2004.
[92] K. M. Chandy and L. Lamport, "Distributed snapshots: determining global states of distributed systems," *ACM Transactions on Computer Systems (TOCS),* vol. 3, pp. 63-75, 1985.
[93] Web-Page. (2008, November). *Provenance Challenge Wik*. Available: http://twiki.ipaw.info/bin/view/Challenge/WebHome
[94] L. Moreau, B. Ludäscher, I. Altintas, R. S. Barga, S. Bowers, S. Callahan*, et al.*, "Special issue: The first provenance challenge," *Concurrency and computation: practice and experience,* vol. 20, pp. 409-418, 2008.
[95] J. Zhao, C. Wroe, C. Goble, R. Stevens, D. Quan, and M. Greenwood, "Using semantic web technologies for representing e-science provenance," in The Semantic Web (ISWC). vol. 3298, S. McIlraith, D. Plexousakis, and F. van Harmelen, Eds., Berlin, Germany: Springer, 2004, pp. 92-106.
[96] Web-Page. (2015, March 10). *Pegasus workflow management system*. Available: http://pegasus.isi.edu/
[97] C. Lin, S. Lu, Z. Lai, A. Chebotko, X. Fei, J. Hua*, et al.*, "Service-oriented architecture for VIEW: A visual scientific workflow management system," in Proceeding of *IEEE International Conference on Services Computing (SCC'08)*, Honolulu, Hawaii, 2008, pp. 335-342.
[98] R. Barga, J. Jackson, N. Araujo, D. Guo, N. Gautam, and Y. Simmhan, "The Trident scientific workflow workbench," in Proceeding of *IEEE Fourth International Conference on eScience (eScience '08)*, Indianapolis, Indiana, 2008, pp. 317-318.
[99] U. Braun, S. Garfinkel, D. A. Holland, K.-K. Muniswamy-Reddy, and M. I. Seltzer, "Issues in automatic provenance collection," in Provenance and annotation of data, L. Moreau and I. Foster, Eds., Berlin, Germany: Springer, 2006, pp. 171-183.
[100] P. M. Kelly, "Applying functional programming theory to the design of workflow engines," Ph.D. dissertation, Department of Computer Science, University of Adelaide, Adelaide, South Australia, Australia, 2011.
[101] D. Hollingsworth, "Workflow management coalition the workflow reference model," Workflow Management Coalition(WfMC), Hampshire, United Kingdom, Technical Report: TC00-1003, 1995.

[102] M. Purvis, M. Purvis, and S. Lemalu, "A framework for distributed workflow systems," in Proceeding of *Hawaii International Conference on System Sciences (HICSS)*, Hawaii, 2001.
[103] M. Sonntag, D. Karastoyanova, and E. Deelman, "Bridging the Gap between Business and Scientific Workflows: Humans in the Loop of Scientific Workflows," in Proceeding, 2010, pp. 206-213.
[104] U. Yildiz, A. Guabtni, and A. H. Ngu, "Business versus scientific workflows: A comparative study," in Proceeding of *Services-I, 2009 World Conference on*, 2009, pp. 340-343.
[105] B. Ludascher, M. Weske, T. McPhillips, and S. Bowers, "Scientific workflows: Business as usual?," *Business Process Management,* pp. 31-47, 2009.
[106] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil*, et al.*, "Pegasus: Mapping scientific workflows onto the grid," in Proceeding of *Grid Computing*, 2004, pp. 11-20.
[107] J. Yu, R. Buyya, and C. K. Tham, "Cost-based scheduling of scientific workflow applications on utility grids," in Proceeding of *First International Conference on e-Science and Grid Computing (e-Science 2005)*, Melbourne, Victoria, Australia, 2005.
[108] Web-Page. (2010, October). *Kepler User Manual (Version 2.1.0 ed.)*. Available: https://code.kepler-project.org/code/kepler-docs/trunk/outreach/documentation/shipping/2.1/UserManual.pdf
[109] A. Colyer, A. Rashid, and G. Blair, "On the separation of concerns in program families," Computing Department, Lancaster University, Lancashire, United Kingdom, Technical Report: COMP-001-2004, 2004.
[110] W. L. Hürsch and C. V. Lopes, "Separation of concerns," Northeastern University, Boston, Massachusetts, Technical Report: NU-CCS-95-03, 1995.
[111] D. Webb and A. Wendelborn, "The PAGIS Grid Application Environment," in Computational Science — ICCS 2003. vol. 2659, P. A. Sloot, D. Abramson, A. Bogdanov, Y. Gorbachev, J. Dongarra, and A. Zomaya, Eds., Berlin, Germany Springer, 2003, pp. 1113-1122.
[112] F. Eliassen, A. Andersen, G. S. Blair, F. Costa, G. Coulson, V. Goebel*, et al.*, "Next generation middleware: Requirements, architecture, and prototypes," in Proceeding of *Workshop on Future Trends of Distributed Computing Systems (FTDCS)*, Cape Town, South Africa, 1999, pp. 60-65.
[113] G. S. Blair, G. Coulson, P. Robin, and M. Papathomas, "An architecture for next generation middleware," in Proceeding of *the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, The Lake District, United Kingdom, 1998, pp. 191-206.
[114] Web-Page. (2015, March 30). *Reification (computer science)*. Available: http://en.wikipedia.org/wiki/Reification_(computer_science)
[115] R. Barbosa and L. M. Pinho, "Mechanisms for reflection-based monitoring of real-time systems," School of Engineering, Polytechnic Institute of Porto, Porto, Portugal, Technical Report: HURRAY-TR-0420, 2004.
[116] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The case for reflective middleware," *Communications of the ACM,* vol. 45, pp. 33-38, 2002.
[117] G. Kiczales, "Towards a new model of abstraction in the engineering of software," in Proceeding of *International Workshop on Reflection and Meta-Level Architecture*, 1992, pp. 67-76.
[118] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier*, et al.*, "Aspect-oriented programming," in Object-Oriented Programming (ECOOP'97).

vol. 1241, M. Akşit and S. Matsuoka, Eds., Berlin, Germany Springer, 1997, pp. 220-242.
[119] Y. Yokote, "The Apertos reflective operating system: The concept and its implementation," *ACM SIGPLAN Notices,* vol. 27, pp. 414-434, 1992.
[120] G. Coulson, G. S. Blair, M. Clarke, and N. Parlavantzas, "The design of a configurable and reconfigurable middleware platform," *Distributed Computing,* vol. 15, pp. 109-126, 2002.
[121] G. S. Blair, G. Coulson, and P. Grace, "Research directions in reflective middleware: the Lancaster experience," in Proceeding of *Proceedings of the 3rd workshop on Adaptive and reflective middleware*, Toronto, Ontario, Canada, 2004, pp. 262-267.
[122] G. Coulson, G. Blair, N. Parlavantzas, W. K. Yeung, and W. Cai, "Applying the reflective middleware approach in grid computing," *Concurrency and Computation: Practice and Experience,* vol. 16, pp. 433-440, 2004.
[123] B. Foote, "Object-Oriented Reflective Metalevel Architectures: Pyrite or Panacea?," in Proceeding of *Workshop on Reflection and Metalevel Architectures (ECOOP/OOPSLA '90)*, Ottawa, Canada, 1990.
[124] D. Edmond and A. H. Ter Hofstede, "A reflective infrastructure for workflow adaptability," *Data & Knowledge Engineering,* vol. 34, pp. 271-304, 2000.
[125] F. Estrella, Z. Kovacs, J.-M. Le Goff, R. McClatchey, T. Solomonides, and N. Toth, "Pattern reification as the basis for description-driven systems," *Software and Systems Modeling,* vol. 2, pp. 108-119, 2003.
[126] F. Estrella, Z. Kovacs, J.-M. Le Goff, R. McClatchey, and N. Toth, "Meta-data objects as the basis for system evolution," in Advances in Web-Age Information Management. vol. 2118, X. S. Wang, G. Yu, and H. Lu, Eds., Berlin, Germany Springer, 2001, pp. 390-399.
[127] R. McClatchey, A. Branson, A. Anjum, P. Bloodsworth, I. Habib, K. Munir*, et al.*, "Providing traceability for neuroimaging analyses," *International journal of medical informatics,* vol. 82, pp. 882-894, 2013.
[128] R. McClatchey, J. Le Goff, N. Baker, W. Harris, and Z. Kovacs, "A distributed workflow and product data management application for the construction of large scale scientific apparatus," in Workflow Management Systems and Interoperability. vol. 164, A. Doğaç, L. Kalinichenko, M. T. Özsu, and A. Sheth, Eds., Berlin, Germany Springer, 1998, pp. 18-34.
[129] J. Shamdasani, A. Branson, and R. McClatchey, "Towards Semantic Provenance in CRISTAL," in Proceeding of *the 3rd International Workshop on the role of Semantic Web in Provenance Management (SWPM 2012)*, Heraklion, Crete, Greece, 2012, pp. 29-36.
[130] S. A. Neuendorffer, "Actor-oriented metaprogramming," Ph.D. dissertation, Department of Electrical Engineering and Computer Sciences, Univeristy of California, Berkeley, California, 2004.
[131] S. Neuendorffer, "Automatic Specialization of Actor-Oriented Models in Ptolemy II," M.S. thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, 2002.
[132] D. Webb, A. L. Wendelborn, and J. Vayssiere, "A study of computational reconfiguration in a process network," in Proceeding of *Integrated Data Environments - Australia (IDEA7)*, Victor Harbour, South Australia, Australia, 2000.
[133] K. Lieberherr and D. H. Lorenz, "Coupling aspect-oriented and adaptive programming," in Aspect-Oriented Software Development, R. E. Filman, T. Elrad, S. Clarke, and M. Aksit, Eds.: Addison-Wesley Professional, 2004, pp. 145-164.

[134] G. T. Sullivan, "Aspect-oriented programming using reflection and metaobject protocols," *Communications of the ACM,* vol. 44, pp. 95-97, 2001.
[135] P. Grace, E. Truyen, B. Lagaisse, and W. Joosen, "The case for aspect-oriented reflective middleware," presented at the the 6th international workshop on Adaptive and reflective middleware: held at the ACM/IFIP/USENIX International Middleware Conference, Newport Beach, California, 2007.
[136] R. Filman, T. Elrad, S. Clarke, and M. Aksit, *Aspect-oriented software development*: Addison-Wesley Professional, 2004.
[137] R. Johnson, J. Hoeller, A. Arendsen, C. Sampaleanu, R. Harrop, T. Risberg, *et al.* (2015, March 10). *Aspect Oriented Programming with Spring*. Available: http://static.springsource.org/spring/docs/2.5.5/reference/aop.html
[138] E. A. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE,* vol. 83, pp. 773-801, 1995.
[139] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token dataflow architecture," *Computers, IEEE Transactions on,* vol. 39, pp. 300-318, 1990.
[140] Arvind and D. E. Culler, "Dataflow architectures," *Annual review of computer science,* vol. 1, pp. 225-253, 1986.
[141] Web-Page. (2015, March 10). *The Kepler Project*. Available: https://kepler-project.org/
[142] M. K. Anand, "Managing Scientific Workflow Provenance," Ph.D. dissertation, Department of Computer Science, Univeristy of California Davis, Davis, California, 2010.
[143] S. Bowers and B. Ludäscher, "Actor-oriented design of scientific workflows," in Conceptual Modeling (ER 05). vol. 3716, L. Delcambre, C. Kop, H. Mayr, J. Mylopoulos, and O. Pastor, Eds., Berlin, Germany Springer, 2005, pp. 369-384.
[144] Web-Page. (2013, April 10). *getting started with kepler provenance 2.4*. Available: https://code.kepler-project.org/code/kepler/trunk/modules/provenance/docs/provenance.pdf
[145] G. Kahn, "The semantics of a simple language for parallel programming," in Proceeding of *the IFIP Congress*, Stockholm, Sweden, 1974, pp. 471-475.
[146] J. Vayssiere, D. Webb, and A. Wendelborn, "An Object-Oriented API for Process Networks," Department of Computer Science, University of Adelaide, Adelaide, Australia, Technical Report: TR 99-03, 1999.
[147] G. Kahn and D. MacQueen, "Coroutines and networks of parallel processes," in Proceeding of *the IFIP Congress*, Toronto, Canada, 1976.
[148] T. M. Parks, "Bounded scheduling of process networks," Ph.D. dissertation, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, 1995.
[149] M. Goel, "Process Networks in Ptolemy II," M.S. thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, California, 1998.
[150] X. Yuhong, E. Lee, L. Xiaojun, Z. Yang, and L. C. Zhong, "The design and application of structured types in Ptolemy II," in Proceeding of *IEEE International Conference on Granular Computing*, Beijing, China, 2005, pp. 683-688.
[151] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, H. Zheng, *et al.*, "Heterogeneous concurrent modeling and design in java (volume 2: Ptolemy ii software architecture)," EECS Department, University of California, Berkley, California, Technical Report: UCB/EECS-2008-29-VOL-2 2008.

[152] S. Bowers, T. M. McPhillips, and B. Ludäscher, "Provenance in collection-oriented scientific workflows," *Concurrency and Computation: Practice and Experience,* vol. 20, pp. 519-529, 2008.
[153] L. Moreau, J. Freire, J. Futrelle, R. McGrath, J. Myers, and P. Paulson, "The open provenance model: An overview," in Provenance and Annotation of Data and Processes. vol. 5272, J. Freire, D. Koop, and L. Moreau, Eds., Berlin, Germany: Springer, 2008, pp. 323-326.
[154] L. Ding, J. Michaelis, J. McCusker, and D. L. McGuinness, "Linked provenance data: A semantic Web-based approach to interoperable workflow traces," *Future Generation Computer Systems,* vol. 27, pp. 797-805, 2011.
[155] D. P. Lanter, "Design of a lineage-based meta-data base for GIS," *Cartography and Geographic Information Systems,* vol. 18, pp. 255-261, 1991.
[156] T. Ellqvist, "Supporting Scientific Collaboration through Workflows and Provenance," Ph.D. dissertation, Department of Computer and Information Science, Linköping University, Linköping, Sweden, 2010.
[157] N. Kwasnikowska and J. Van den Bussche, "Mapping the NRC dataflow model to the open provenance model," in Provenance and Annotation of Data and Processes, J. Freire, D. Koop, and L. Moreau, Eds., Berlin, Germany: Springer, 2008, pp. 3-16.
[158] D. Garijo and Y. Gil. (2012). *Towards Open Publication of Reusable Scientific Workflows: Abstractions, Standards and Linked Data*. Available: http://www.isi.edu/~gil/papers/garijo-gil-opmw12.pdf
[159] N. Kwasnikowska, L. Moreau, and J. Van den Bussche, "A formal account of the open provenance model," *Transactions on the Web, to be published,* 2015.
[160] P. Bourne, L. Xie, and S. Kinnings. (2015, Feburary 06). *TB-drugome*. Available: http://www.wings-workflows.org/drugome/index.php/Main_Page
[161] Web-Page. (2015, January 06). *WINGS*. Available: http://www.wings-workflows.org/
[162] P. Groth, M. Luck, and L. Moreau, "A protocol for recording provenance in service-oriented grids," in Principles of Distributed Systems, Berlin, Germany: Springer-Verlag 2005, pp. 124-139.
[163] Y. Simmhan and R. Barga, "Analysis of approaches for supporting the Open Provenance Model: A case study of the Trident workflow workbench," *Future Generation Computer Systems,* vol. 27, pp. 790-796, 2011.
[164] C. Lim, S. Lu, A. Chebotko, and F. Fotouhi, "Storing, reasoning, and querying OPM-compliant scientific workflow provenance using relational databases," *Future Generation Computer Systems,* vol. 27, pp. 781-789, 2011.
[165] C. Lim, S. Lu, A. Chebotko, and F. Fotouhi, "Prospective and retrospective provenance collection in scientific workflow environments," in Proceeding of *International Conference on Services Computing (SCC)*, Miami, Florida, 2010, pp. 449-456.
[166] P. Buneman and W.-C. Tan, "Provenance in databases," in Proceeding of *ACM SIGMOD international conference on Management of data*, Beijing, China, 2007, pp. 1171-1173.
[167] S. Bowers, T. McPhillips, and B. Ludäscher, "Declarative rules for inferring fine-grained data provenance from scientific workflow execution traces," in Provenance and Annotation of Data and Processes, P. Groth and J. Frew, Eds., Berlin, Germany: Springer, 2012, pp. 82-96.
[168] T. Malik, L. Nistor, and A. Gehani, "Tracking and sketching distributed data provenance," in Proceeding of *Sixth International Conference on e-Science (e-Science 2010)*, Brisbane, Queensland, Australia, 2010, pp. 190-197.

[169] B. Glavic, K. S. Esmaili, P. M. Fischer, and N. Tatbul, "The Case for Fine-Grained Stream Provenance," in Proceeding of *Conference on Datenbanksysteme in Buisness, Technologie und Web (BTW) Workshops*, Kaiserslautern , Germany, 2011, pp. 58-61.
[170] B. Glavic, K. Sheykh Esmaili, P. M. Fischer, and N. Tatbul, "Ariadne: Managing fine-grained provenance on data streams," in Proceeding of *the 7th ACM international conference on Distributed event-based systems*, Arlington, Texas, 2013, pp. 39-50.
[171] D. Tariq, M. Ali, and A. Gehani, "Towards Automated Collection of Application-Level Data Provenance," in Proceeding of *Theory and Practice of Provenance (TaPP'12)*, Boston, Massachusetts, 2012.
[172] M. Sonntag, D. Karastoyanova, and E. Deelman, "Bridging the Gap between Business and Scientific Workflows: Humans in the Loop of Scientific Workflows," in Proceeding of *Sixth International Conference on e-Science (e-Science 2010)*, Brisbane, Queensland, Australia, 2010, pp. 206-213.
[173] U. Yildiz, A. Guabtni, and A. H. Ngu, "Business versus scientific workflows: A comparative study," in Proceeding of *International Conference on Services Computing (SCC)*, Bangalore, India, 2009, pp. 340-343.
[174] B. Ludäscher, M. Weske, T. McPhillips, and S. Bowers, "Scientific workflows: Business as usual?," in Business Process Management. vol. 5701, U. Dayal, J. Eder, J. Koehler, and H. Reijers, Eds., Berlin, Germany: Springer, 2009, pp. 31-47.
[175] Web-Page. (2008, August 18). *Inferences*. Available: http://twiki.ipaw.info/bin/view/Challenge/OPM1-01Review-Inferences
[176] E. Deelman, D. Gannon, M. Shields, and I. Taylor, "Workflows and e-Science: An overview of workflow system features and capabilities," *Future Generation Computer Systems,* vol. 25, pp. 528-540, 2009.
[177] R. Stevens, J. Zhao, and C. Goble, "Using provenance to manage knowledge of in silico experiments," *Briefings in bioinformatics,* vol. 8, pp. 183-194, 2007.
[178] A. Marinho, C. Werner, S. da Cruz, M. Mattoso, V. Braganholo, and L. Murta, "A strategy for provenance gathering in distributed scientific workflows," in Proceeding of *International Conference on Services Computing (SCC)*, Bangalore, India, 2009, pp. 344-347.
[179] Web-Page. (2010, July 12). *The Fourth and Last Provenance Challenge*. Available: http://twiki.ipaw.info/bin/view/Challenge/FourthProvenanceChallenge
[180] Web-Page. (2014,July 05). *Lineage File System*. Available: http://crypto.stanford.edu/~cao/lineage.html
[181] R. P. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok, "Story Book: An Efficient Extensible Provenance Framework," in Proceeding of *Theory and Practice of Provenance (TaPP'09)*, San Francisco, California, 2009.
[182] T. Malik, A. Gehani, D. Tariq, and F. Zaffar, "Sketching Distributed Data Provenance," in Data Provenance and Data Management in eScience, Q. Liu, Q. Bai, S. Giugni, D. Williamson, and J. Taylor, Eds., Berlin, Germany: Springer, 2013, pp. 85-107.
[183] A. Gehani and D. Tariq, "SPADE: Support for provenance auditing in distributed environments," in Middleware 2012. vol. 7662, P. Narasimhan and P. Triantafillou, Eds., Berlin, Germany: Springer, 2012, pp. 101-120.
[184] I. T. Foster, J.-S. Vöckler, M. Wilde, and Y. Zhao, "The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration," in Proceeding of *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, California, 2003, pp. 18-29.
[185] Web-Page. (2014, July 10). *Provenance Aware Service Oriented Architecture (PASOA)*. Available: http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/WebHome

[186] B. Cao, B. Plale, G. Subramanian, E. Robertson, and Y. Simmhan, "Provenance information model of karma version 3," in Proceeding of *International Conference on Services Computing (SCC)*, Bangalore, India, 2009, pp. 348-351.
[187] Y. L. Simmhan, B. Plale, and D. Gannon, "Karma2: Provenance management for data-driven workflows," *International Journal of Web Services Research,* vol. 5, pp. 1-22, 2008.
[188] A. Heydon, R. Levin, T. Mann, and Y. Yu, "The Vesta approach to software configuration management," Compaq Systems Research Center, California Technical Report: 1999-001, 1999.
[189] C. Otero, *Software engineering design: theory and practice*, 1st ed. Boca Raton, Florida: CRC Press, 2012.
[190] M. S. Aktas, B. Plale, D. Leake, and N. K. Mukhi, "Unmanaged Workflows: Their Provenance and Use," in Data Provenance and Data Management in eScience, Q. Liu, Q. Bai, S. Giugni, D. Williamson, and J. Taylor, Eds., Berlin, Germany: Springer, 2013, pp. 59-81.
[191] T. De Nies, S. Coppens, D. Van Deursen, E. Mannens, and R. Van de Walle, "Automatic discovery of high-level provenance using semantic similarity," in Provenance and Annotation of Data and Processes, P. Groth and J. Frew, Eds., Berlin, Germany: Springer, 2012, pp. 97-110.
[192] S. Magliacane, "Reconstructing provenance," in The Semantic Web (ISWC). vol. 7650, P. Cudré-Mauroux, J. Heflin, E. Sirin, T. Tudorache, J. Euzenat, M. Hauswirth, *et al.*, Eds., Berlin, Germany: Springer, 2012, pp. 399-406.
[193] E. Freeman, E. Robson, B. Bates, and K. Sierra, *Head first design patterns*: O'Reilly Media, Inc., 2004.
[194] Y. Simmhan, B. Plale, D. Gannon, and S. Marru, "Performance Evaluation of the Karma Provenance Framework for Scientific Workflows," in Provenance and Annotation of Data. vol. 4145, L. Moreau and I. Foster, Eds., Berlin, Germany: Springer, 2006, pp. 222-236.
[195] P. Macko and M. Seltzer, "Provenance map orbiter: Interactive exploration of large provenance graphs," in Proceeding of *Theory and Practice of Provenance (TaPP'11)*, Heraklion, Crete, Greece, 2011.
[196] M. A. Borkin, C. S. Yeh, M. Boyd, P. Macko, K. Z. Gajos, M. Seltzer, *et al.*, "Evaluation of filesystem provenance visualization tools," *Visualization and Computer Graphics, IEEE Transactions on,* vol. 19, pp. 2476-2485, 2013.
[197] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. Maclean, D. Margo, *et al.*, "Layering in provenance systems," in Proceeding of *USENIX Annual Technical Conference*, San Diego, California, 2009.
[198] B. Plale, B. Cao, and M. S. Aktas. (2011, January). *Provenance Capture of Unmanaged Workflows with Karma*. Available: http://www.cs.indiana.edu/~plale/papers/Plale-FGCS-preprintJan2011.pdf
[199] A. Chebotko, S. Lu, X. Fei, and F. Fotouhi, "RDFProv: A relational RDF store for querying and managing scientific workflow provenance," *Data & Knowledge Engineering,* vol. 69, pp. 836-865, 2010.
[200] M. E. Research. (2008). *Trident Workbench: A Scientific Workflow System*. Available: http://research.microsoft.com/en-us/collaboration/tools/trident.aspx
[201] M. D. Valerio, S. S. Sahoo, R. S. Barga, and J. J. Jackson, "Capturing workflow event data for monitoring, performance analysis, and management of scientific workflows," in Proceeding of *IEEE Fourth International Conference on eScience (eScience '08)*, Indianapolis, Indiana, 2008, pp. 626-633.

[202] N. Araujo, R. Barga, D. Guo, J. Jackson, Y. Simmhan, C. van Ingen, *et al.* (2015, March 10). *Trident Scientific Workflow Workbench*. Available: http://research.microsoft.com/en-us/collaboration/tools/tridentescience08tutorial.ppt

[203] L. Ramakrishnan and D. Gannon, "A survey of distributed workflow characteristics and resource requirements," Indiana University, Bloomington, Indiana, Technical Report: TR671, 2008.

[204] S. da Cruz, F. Chirigati, R. Dahis, M. Campos, and M. Mattoso, "Using explicit control processes in distributed workflows to gather provenance," in Provenance and Annotation of Data and Processes. vol. 5272, J. Freire, D. Koop, and L. Moreau, Eds., Berlin, Germany Springer, 2008, pp. 186-199.

[205] M. Sarikhani, B. Javadi, and A. Parichehre, "Fault-aware scheduling in Grid environment based on linear programming," in Proceeding of *International Conference on High Performance Computing and Simulation (HPCS)*, 2010, pp. 656-665.

[206] A. Marinho, L. Murta, C. Werner, V. Braganholo, S. M. S. da Cruz, E. Ogasawara, *et al.*, "Managing Provenance in Scientific Workflows with ProvManager," in Proceeding of *International Workshop on Challenges in e-Science (CIS2010)*, Petrópolis, Rio de Janeiro, Brazil, 2010, pp. 17-24.

[207] S. Chiba, "Javassist-a reflection-based programming wizard for java," in Proceeding of *Object-oriented programming systems, languages and applications (OOPSLA) Workshop on Reflective Programming in C++ and Java*, Vancouver, British Columbia, Canada, 1998.

[208] S. Chiba, "Load-time structural reflection in Java," in Proceeding of *European Conference on Object-Oriented Programming (ECOOP)*, Sophia Antipolis and Cannes, France, 2000, pp. 313-336.

[209] P. David, T. Ledoux, and N. M. Bouraqadi-Saadani, "Two-step weaving with reflection using AspectJ," in Proceeding of *Object-oriented programming systems, languages and applications (OOPSLA) Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.