1-1-2010

Scientific Workflow Integration For Services Computing

Cui Lin
Wayne State University,

SCIENTIFIC WORKFLOW INTEGRATION FOR SERVICES COMPUTING

by

CUI LIN

DISSERTATION

Submitted to the Graduate School

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

2010

MAJOR: COMPUTER SCIENCE

Approved by:

Advisor Date

ACKNOWLEDGMENTS

First of all, I owe my deepest gratitude to my advisor, Professor Shiyong Lu, for his advice, encouragement, and support over the past years. He has spent tremendous time and efforts to help me achieve excellence in both research and teaching. I have also benefited greatly from his rigorous research attitude and his insight on our research.

I am also heartily thankful to my co-advisor, Professor Jing Hua, and my research collaborator, Professor Otto Muzik. I feel enormously fortunate to have access to their powerful intellectual minds and remarkable insights on their research. Their enthusiasm for science is extremely contagious and has deeply influenced me. Furthermore, I would like to express my sincere appreciation to Professors Farshad Fotouhi and Zaki Malik for their valuable advice and efforts as the members of my dissertation committee.

It is a pleasure to thank all the members in the Scientific Workflow Research Laboratory, who have helped me in many aspects over these years. Also, I want to extend my gratitude to all the members in the Graphics and Imaging Laboratory, especially Zhaoqiang Lai and Darshan Pai, for their great support and close research collaborations.

Moreover, I am grateful for the Michigan Technology Tri-Corridor basic research grant MTTC05-135 / GR686 and the Thomas C. Rumble Fellowship Award, which partially supported my research.

Last but not the least, I would like to thank my mother Ruichun Cui and my father Guangyue Lin for their endless love and patience throughout all stages of my growth. Their open minds and encouragement allow me to transit from my industrial career path to the pursuit of success in an academic career. I also want to thank my boy friend, Dr. Yunhao Tan, for his love and support in the past years, which helped me overcome many difficulties and go through some hard time in these years. Without his support, I could not complete this dissertation.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Scientific Workflows and Scientific Workflow Management Systems

Workflow technologies originated from the development of office automation and process re-engineering. In general, a *workflow* is a computerized model of a process, which consists of a collection of interconnected tasks (or called activities, actions, actors) that are constructed to achieve a predefined objective. Workflow concepts and technologies have been developed in the business world for more than two decades, and a *business workflow* is a computerized business process, in which documents, information, or tasks are passed between participants according to defined sets of rules to achieve, or contribute to, an overall business goal [1]. A *business workflow management system* (BWFMS) is a system that provides tools to design, manage, and execute business workflows.

In recent years, significant scientific advances are increasingly achieved through complex scientific processes, which may involve tremendous steps of computations and vast amounts of scientific data for these computations. As such complex computations and data analysis require tremendous amounts of human efforts and manual coordination, to sustain current growth in scientific computations and data, workflows have been applied in scientific domains to automate large-scale scientific processes, termed *scientific workflows*. A *scientific workflow* is a computerized model of a scientific process, in whole or part, which streamlines a collection of tasks with data channels and dataflow constructs to automate data computation and analysis. Scientific workflows have recently emerged as a new paradigm for scientists to integrate, structure, and orchestrate a wide range of analytical tools into complex scientific processes to accelerate scientific discoveries. A *scientific workflow management system* (SWFMS) is the system that completely defines, modifies, manages, monitors, and executes scientific workflows through the execution of scientific tasks whose execution order is driven by a computerized representation of the workflow logic. An SWFMS automates the whole lifecycle of scientific research for scientists, from data collection, hypothesis formation through compute-intensive and data-intensive analysis to the publication and dissemination of scientific results, supporting scientific reproducibility, publication, and sharing.

Since the history of business workflows is far longer than that of scientific workflows, some concepts and technologies developed in the business domain can be migrated to the scientific domain. However, scientific workflows have their domain specific requirements and challenges, many of the techniques developed for BWFMSs cannot be directly applied in SWFMSs. For example, SWFMSs focus on dataflow design patterns and execution models while BWFMSs tend to have controlflow patterns and events. Such differences determine the underlying architecture design and workflow formalism [2] for a workflow system; Scientific workflows are often designed in an *ad hoc* manner and executed in a trial-and-error pattern, while business workflows are usually predefined and executed in a routine fashion. Therefore, the ability to revise, pause, resume, and record workflows required in SWFMSs is not exposed in most BWFMSs.

1.2 Problem Statement

Today, research collaborations are becoming globally dispersed, scientists increasingly rely on the Web technology to perform their *in silico* experiments. These experiments are motivated by different hypotheses formulated for the research of a particular domain, and each of the hypotheses may involve various processes and sub processes. As shown in Figure 1.1, a scientific workflow, as a computerized model of a scientific process or sub process, relies on two types of resources: one is a large number of *analytical tool resources* that are involved in different scientific processes; another is the *compute resources* that are distributed over the network. These analytical tools are often developed as public services or proprietary applications, from different organizations. More and more compute resources are recently exposed as services, which can be directly accessed via a standardized interface through the Internet. A scientific workflow today may comprise hundreds or even thousands such heterogeneous analytic tools, compute services for the execution of these tools may be distributed across different services computing environments, connected by the Internet, so the integration and management of such workflows in SWFMSs are pushing the limit of current scientific workflow technology.

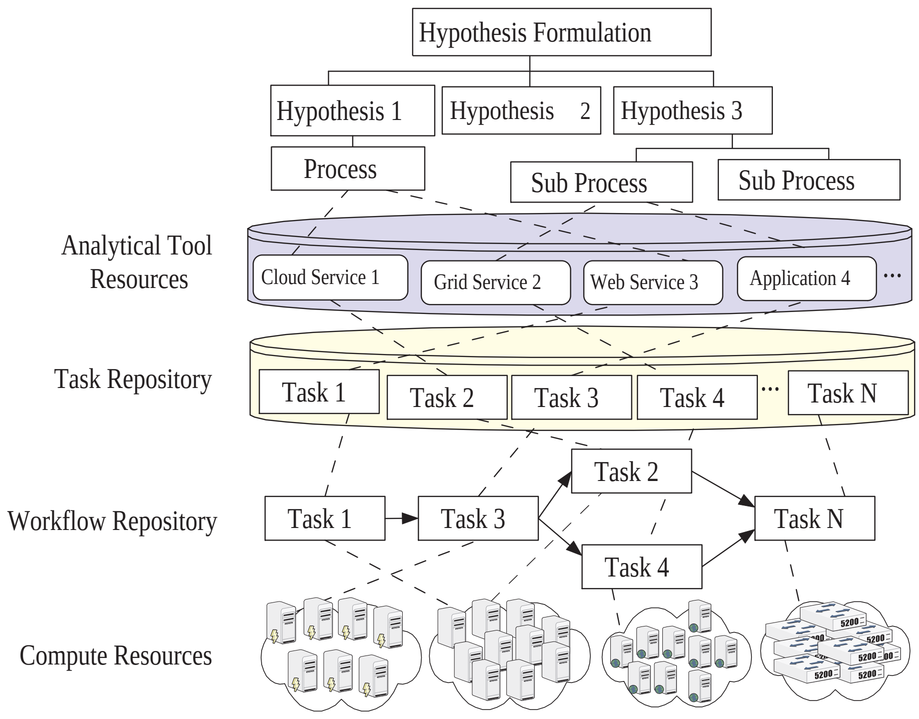


Figure 1.1: An integrated framework to integrate heterogeneous analytical tools and execute them on distributed compute resources.

To solve this problem, my dissertation explores new techniques to build an SWFMS that provides an integrated framework to (1) integrate heterogeneous tools from the analytical tool resources into uniform workflow tasks, and manage them in a *task repository*; (2) compose tasks from the task repository into various scientific workflows and manage them in a *workflow repository*; (3) schedule and execute workflows from the workflow repository in services computing environments. Specifically, this dissertation focuses on the following research issues:

How to design a proper foundation for workflow composition, scheduling, execution and management, so that scientific process automation can be managed systematically?

A fundamental problem missing in current scientific workflow research is a proper foundation, which can be used for the systematic development of scientific workflow systems. The availability of such a reference architecture can not only provide a guidance for the architectural design of an SWFMS, but also provide a proper foundation to integrate all the functions and components in an SWFMS, so that scientific process automation can be managed systematically. While several SWFMSs [3, 4, 5, 6, 7, 8] have been developed during the past few years, an architectural reference that can provide a high-level organization of subsystems and their interactions in an SWFMS is still missing. Because of that, the development of a scientific workflow system is mostly *ad hoc* in scientific workflow design, specification, development, execution, and provenance tracking, etc.

Although the reference architecture proposed by the Workflow Management Coalition (WfMC) and its variant [9] have been widely adopted in the development of different BWFMSs [10, 11, 12, 13, 14], it is not suitable for SWFMSs as business workflows and scientific workflows have different goals. The goal of business workflows is to reduce human resources (and other costs) and increase revenue, while the goal of scientific workflows is to reduce both human and computation costs and accelerate the speed of turning large amounts of bits and bytes into knowledge and discovery. Moreover, business workflows are typically controlflow oriented, while scientific workflows tend to be dataflow oriented, introducing a new set of requirements and challenges for system development, from the support of intensive user-interaction and visualization, customizable and extensible GUI, reproducibility, high-end computing, to heterogeneous data, software tool, and service management.

How to provide an appropriate abstraction to integrate heterogeneous analytical tools, so that they can be composed into various scientific workflows?

In scientific workflows, analytical tools are often developed by various research organizations, originally as software tools for their own scientific problems and then are published for the purpose of sharing and reusing them in solving other scientific problems. Some of these tools are exposed as services, such as Cloud services, Grid services or Web services; some of them are developed as proprietary applications. Therefore, it is very common that these tools are written in various programming languages, invoked via different invocation mechanisms, and run in disparate computing environments.

The techniques that integrate heterogeneous programs into business workflows cannot be directly applied to scientific workflows due to the fundamental differences between business workflows and scientific workflows. More specifically, their approaches cannot be used to abstract heterogeneous and distributed services and applications into uniform dataflow-based scientific workflow tasks (i.e., tasks with well-defined input and output ports). Some existing SWFMSs provide built-in system-specific wrappers for the invocation of external services and applications; however, they do not support the flexible mappings between the input/output ports of a task interface and the inputs/outputs of a task component. In most cases, such mappings have to be performed by the development of a custom task inside an SWFMS, and the task wraps the invocation of an external tool and hardcodes the mappings between inputs/outputs of the external tool and input/output ports of the task. This wrapper-programming approach is not only unnecessarily tedious, error-prone, but also lacks the flexibility of supporting multiple alternative task components and the dynamic binding capability between a task interface and a task component. Therefore, how to abstract heterogeneous analytical tools into uniform workflow tasks remains an open research problem.

The shimming problem is another workflow integration problem caused by the heterogeneity. As most of third-party services and applications are syntactically mismatching or semantically incompatible, a special kind of workflow components, called *shims*, is used to mediate them. A shim takes the output data of an upstream workflow task, performs some transformation, and then feeds the data to the input of a downstream task. The shimming problem has been widely recognized as an important problem in the community [15, 16], leading to much efforts in the development of shims [17], shim-aware workflow composition [15], and the suggestion of a new discipline called *shimology* [16]. Existing shimming techniques have two serious limitations. First, they produce scientific workflows that are cluttered with many visible shims. Ideally, these shims should be hidden from scientists so that they can better focus on functional components of workflows. Second, these techniques still require a user to write custom wrapper around a task component according to the task programming model of a system. Moreover, these hard-coded implicit shims are irreusable across other tasks. Therefore, how to provide a flexible mapping between task ports and inputs/outputs of task components poses another challenge for workflow integration.

How to provide a run-time framework to schedule and execute workflows on distributed compute resources in services computing environments?

In a services computing environment, compute resources for a workflow can be assigned by service requests, and each workflow can be assigned different number and type of resources for executions. The compute resources assigned to a workflow can be *elastically* scaled out or scaled in on demand of the size of the workflow. The execution time of each task in a workflow may differ using different resources and data associated with a task are required to be transferred from one resource to another with different data transfer rates between resources. Therefore, how to schedule a workflow onto suitable compute resources, so that the execution of a workflow can be completed with the satisfaction of a predefined objective function is one of the key problems in workflow management.

Generally, the workflow scheduling problem is known as an NP-complete problem [18, 19].

Many heuristics and guided random search based algorithms have been proposed in the literature [20]; however, none of the algorithms provides a solution to schedule a workflow in a services computing environment, in which resources assigned to a workflow can be elastically scaled out or scaled in at runtime. This motivates our research in this direction. Furthermore, once a task is scheduled onto a compute resource, how to provide a runtime framework to manage, execute, and monitor tasks and their associated data in heterogeneous and distributed computing environments is another challenging research problem.

1.3 Contributions

To solve the above problems in scientific workflow integration, in this dissertation, we propose an integrated solution for composing, scheduling, executing, and developing scientific workflows and scientific workflow management systems. Furthermore, we have developed a service oriented workflow management system, the VIEW system, and a VIEW based workflow application system, the FiberFlow system, to validate our architectures, models, languages, and algorithms. Specifically, the contributions of this dissertation are as follows:

1. **Reference Architecture for Scientific Workflow Systems** (Chapter 3). We identify seven key architectural requirements that are particularly needed by scientific workflow systems. In response to these requirements, we propose the *first* reference architecture for scientific workflow management systems, which is composed of four logical layers, seven major functional subsystems, and six interfaces. The reference architecture not only provides a high-level organization of subsystems and their interactions in a workflow system, but also provides a basis for comparison between different systems and a guidance for the architectural design of an SWFMS in a specific scientific domain.

2. **Task Template Model and Task Specification Language** (Chapter 4). We propose a task template model which not only provides an appropriate abstraction of heterogeneous

services and applications, but also encapsulates the composition and mapping of shims and functional task components within a task interface. We design an XML-based task specification language, called TSL, to realize the proposed task template model. TSL not only enables the abstraction of heterogeneous services and applications into uniform workflow tasks, but also provides a solution to address both TYPE-I and TYPE-II shimming problems in composing scientific workflows. To our best knowledge, this is the *first* shimming technique that makes shims invisible at the workflow level, resulting in scientific workflows that are more elegant and readable.

3. **Task Run Model and Task Run Description Language** (Chapter 4). We propose a task run model to model the run-time behaviors of tasks. Based on the task run model, we design the task run description language, TRDL, for the description of task runs, enabling the execution of task instances constructed from heterogeneous services and applications. Furthermore, we propose a service-oriented architecture for task management to enable the integration of heterogeneous services and applications into scientific workflows in distributed environments. Our proposed models, languages, and architecture provide a programming language and platform independent framework; they are extensible for future services and applications.

4. **Workflow Scheduling for Services Computing Environment** (Chapter 5). We identify the workflow scheduling problem for a services computing environment, in which resources assigned to a workflow can be *elastically* scaled out or scaled in on demand of the size of the workflow. We firstly formalize a workflow cost model that enables the prediction and estimation of workflow execution. Based on the proposed cost model, two novel workflow scheduling algorithms, the SHEFT (Scalable-Earliest-Finish-Time-First) and the SCPOR (Scalable-Critical-Path-On-a-Resource) algorithm, are proposed to map tasks onto suitable resources and order their executions in services computing environments. Our extensive experiments show that the proposed algorithms outper-

form other competitive algorithms especially for scalable, compute-intensive and data-intensive workflows.

5. **Visual Scientific Workflow Management System** (Chapter 6). We develop an SOA based VIEW system to validate the proposed architectures, models, languages, and algorithms. VIEW consists of six loosely-coupled service components, each of which corresponds to a functional component that is identified in the reference architecture, whose functionality is exposed as a Web service. We present a service configuration management in the VIEW system that provides a flexible configuration functionality for the VIEW subsystems. Furthermore, we develop a VIEW based scientific workflow application system, called FiberFlow system, to demonstrate the capabilities of the VIEW system in support of user-interaction intensive and visualization-intensive scientific workflows.

1.4 Dissertation Organization

The remainder of the dissertation is organized as follows. In *Chapter 2*, we briefly review prior work done in workflow integration, workflow scheduling and the development of workflow management systems. Then in *Chapter 3* we propose the first reference architecture for scientific workflow management systems, which provides a high-level organization of subsystems and their interactions in an SWFMS. In *Chapter 4* a task model is presented to enable the abstraction of heterogeneous services and applications into uniform workflow tasks, and the proposed task languages based on the task model provide an integrated solution to solve the shimming problems and support the execution of tasks in heterogeneous and distributed computing environments. In *Chapter 5*, our proposed workflow scheduling techniques are presented to enable tasks in a workflow to be scheduled in a services computing environment. In *Chapter 6*, we discuss the system architecture of the developed VIEW system and present a VIEW based workflow application system, called the FiberFlow system. Finally, we conclude this dissertation and outline some future research work in *Chapter 7*.

CHAPTER 2: RELATED WORK

Scientific workflow has been well recognized as a multi-disciplinary research area, in which technologies from various domains have contributed to its development. This chapter introduces the technologies that are most relevant to our solutions proposed in this dissertation. As the system architecture and major functionalities provide a foundation for the development and management of a workflow system, we firstly discuss the architectures of existing workflow management systems proposed in both business and science domains in Section 2.1. Then in Section 2.2, we analyze the state-of-the-art workflow integration techniques applied in both business and scientific workflow systems. Finally, we focus on a comprehensive survey of existing workflow scheduling algorithms developed for a variety of computing environments in Section 2.3.

2.1 Architectures of Workflow Management Systems

A system architecture defines the fundamental organization of a system. Beyond the algorithms and data structures of the computation, It designs and specifies the overall system structure, which comprises system components, the properties of these components, the relationships among them, and the principles governing its design and evolution. The architecture of workflow management systems (WFMS) mainly considers major system components and subsystems and what kinds of interactions between these subsystems. In Section 2.1.1 and Section 2.1.2, we investigate a series of business workflow systems and scientific workflow systems whose architecture designs emphasize the aspects of system distribution and workflow integration.

2.1.1 Architectures of Business Workflow Management Systems

Historically, business workflow management systems can be traced back to office automation systems of the 1970's and 80's, and gained momentum in the 90's under different names including business process modeling and business process reengineering [21, 22, 23, 24, 25, 26, 27, 28, 29, 12, 30, 31]. According to the system distribution model, the representative architectures of business workflow management systems (BWFMS) can be distinguished as the following five variations: centralized-server, replicated-server, localized-server, no-server and service oriented architecture.

Centralized-server architecture. The BWFMS is built upon a centralized workflow enactment server or a centralized data storage system, which has to be accessed for the execution of each task in a workflow. For example, the Process WEAVER system [25] is built around a centralized workflow enactment server and all the communications between servers and other system components are controlled by broadcast message server (BMS) [32]. Although such an architecture supports system extensibility, the major disadvantage of systems with such architecture [22, 23, 24, 26, 28] is the lack of scalability to execute large-scale distributed workflows.

Replicated-server architecture. The BWFMS comprises multiple identical servers and each server has a workflow enactment and a workflow database subsystem, which contains complete information required for workflow execution. For example, the FlowMark system [30] consists of multiple workflow enactment servers, and each of them is connected as a client to a database server. FlowMark servers can reside in remote hosts other than the one where the database server is located through the communication of TCP/IP, NetBIOS or APPC. The distribution of workflow execution is specified in the workflow specification. FlowMark can run across different platforms (AIX, OS/2, Windows); however, persistent data resides in a single database server, which is used to store all workflow execution data. It has facilitated the design of the overall system, but introduces a single point of failure in the architecture. Other systems based on a centralized database may suffer from the same problem [28]. The main issue of this architecture is how to accomplish the efficient replication of workflow information and how to handle the extensive network traffic for distributed workflow execution.

Localized-server architecture. The BWFMS allows multiple servers co-exist, and each of them is localized close to where activity execution takes place [24]. For example, the

METEOR_2 [29, 12] system was developed with the emphasis on scalability and large-scale distributed workflow execution. Three prototype implementations have been developed based on the METEOR_2 model : *OrbWork*, a fully distributed CORBA-based dynamic workflow enactment system [33]; *NeoWork*, a CORBA-based workflow enactment system with centralized schedulers; and *WebWork*, a fully distributed workflow enactment system relying on Web technology, which supports the development of workflow applications that can run in heterogeneous and distributed environments. Each of these workflow enactment systems contains workflow schedulers, task managers, and a run-time monitor. In METEOR_2 and other systems with similar architectures [26, 26], a centralized monitoring server must be always available in an operating workflow system, as every step of workflow execution has to be reported to the monitoring server, in order to perform the system recovery.

Server-less architecture: The BWFMS migrates the server functionality to every participating client in the workflow system. Each workflow run migrates from one client to another during workflow execution, so the state of the workflow execution is distributed over the whole workflow system. A distributed synchronization mechanism is required to determine the actors which execute workflow activities. To support such architecture, the Exotica/FMQM system [28] has a reliable communication system based on persistent message queues used to connect processing nodes. After activity execution is completed, messages are sent to all subsequent processing nodes which are defined in the workflow graph. The system may choose to retrieve data produced from a task from the output queue of the previous node. Node synchronization is thus achieved by means of the transaction system provided by the queuing system. The main problem with this architecture is that of excessive network traffic. As data produced by a task has to be sent to all nodes in the system and it is not known on which eligible node the task will actually execute before node synchronization has taken place.

Service-oriented architecture. More and more recently developed workflow systems are developed on service-oriented architecture [34, 35]. For example, the YAWL system [36, 37] consists of four YAWL services: (1) YAWL worklist handler, (2) YAWL web services broker, (3) YAWL interoperability broker, and (4) custom YAWL services. The *YAWL worklist handler* corresponds to the classical worklist handler present in most workflow management systems. It is the component that is used to assign work to users of the system. Through the worklist handler, users can accept work items and signal their completion. In traditional workflow systems, the *worklist handler* is embedded in the workflow engine; in YAWL, however, it is considered to be a service decoupled from the engine. The *YAWL web services* broker is the glue between the engine and other web services. Note that it is unlikely that web services will be able to directly connect to the YAWL engine, since they will typically be designed for more general purposes than just interacting with a workflow engine; The *YAWL interoperability broker* is a service designed to interconnect different workflow engines. A *custom YAWL service* connects the engine with an entity in the environment of the system.

2.1.2 Architectures of Scientific Workflow Management Systems

Even though BWFMSs have been under investigation for more than 20 years, there is no proper business workflow management system architecture that can be directly adapted to scientific workflow management systems. While several scientific workflow management systems (SWFMS) [3, 4, 5, 6, 7, 8] have been developed during the past few years, which provide much experience for future research and development, an architectural reference that can provide a high-level organization of subsystems and their interactions in an SWFMS is missing. The development of a scientific workflow system is mostly *ad hoc* in scientific workflow design, specification, development, execution, and provenance tracking, etc. We investigate several representative SWFMSs on their architectures, models and unique features.

Actor-oriented. The Kepler system [3, 38, 39, 40, 41] is an open-source scientific workflow workflow management system. A unique feature of the Kepler is its actor-oriented modeling, inherited from the underlying dataflow oriented Ptolemy II system [42], which is to build models of systems based on the assembly of pre-designed components and these components are called *actors* [43]. An actor is an encapsulation of parameterized actions performed on input data to produce output data. An actor may be stateless or statefull, depending on whether it has an internal state. Communication between actors happens through interfaces called *ports*. Each actor has *input ports* and *output ports*. In addition to ports, actors have *parameters*, which configure and customize the actors' behavior. Ports and parameters are the interfaces of an actor. Actors can be regarded as reusable independent blocks of computation and they consume input data from a set of input ports and output results to a set of output ports. A group of actors can be wired together by introducing a mapping from input ports to output ports.

In Ptolemy II, the term of *framework* refers to an environment that actors reside in, and defines the interaction among actors. The interaction styles of actors are captured by *models of computation* (MoC). A MoC defines the communication semantics among ports and the flow of control and data among actors. A framework implements a model of computation. Frameworks and actors together define a system [43]. The Ptolemy system focuses on visual, module-oriented programming with an emphasis on multiple component interaction semantics. It precisely controls the execution model of a workflow via the so-called *directors*, which is the only available workflow management system that allows one to plug in different execution models into workflows [39].

Service-oriented. The Taverna system [44, 45] is an open source, Grid-aware workflow management system, developed for scientists to perform data-intensive *in silico* experiments on distributed resources. Taverna has two major conceptual architectural abstractions: the user perspective and the services perspective, which separates the perspective of user from underlying middleware and operations. A three-tiered data model serves the two abstractions at different levels: In the *application data flow* layer, a user-level workflow object model is applied to present the workflow from a user view, hiding the complexity of the service interactions. Enactor internal object model and myGrid contextual information model are implemented in the *execution flow* layer, which manages data structures and fault recovery on behalf of the user.

This saves users explicitly handling these at the application data flow layer. The *Processor Invocation* layer is used to invoke concrete services using the enactor.

Taverna is implemented as a service oriented architecture. Taverna differs from other SWFMSs by placing an emphasis on coping with an environment of autonomous service providers. Another advanced feature in Taverna is that provenance has become an integral part of the system, which allows scientists to inspect and record their experiment that is composed of local or external services.

Federation-based architecture. The Triana system [46, 47, 48] is an open source, distributed, platform independent, middleware independent problem solving environment [49] and a test application for the GridLab project [50], written in Java. Triana provides a graphical interactive environment that allows users to compose applications and specify their distributed behavior.

One unique feature in Triana lies in its *federated* architecture. It consists of a complex set of interacting components that create the complete system or any subset. In Triana, components and services are aggregated, integrated as execution *Units* or *Group Units* with defined interactions, so users visually interact with units that can be connected and create a workflow by dragging the desired units onto the workspace regardless of their underlying implementation. This federation based architecture gives Triana the flexibility to be applied to many different scenarios and at many levels, and allows to distribute sections of a workflow to remote machines through a connected peer-to-peer network.

From Grid to Service. The Pegasus system [6, 51, 52, 53] provides a framework which maps complex scientific workflows onto distributed Grid resources. Pegasus targets at the computation-intensive workflows, Grid-based workflows and large-scale workflows, which are composed of hundreds of or even thousands of individual tasks or collaborative applications. The Pagasus system can run workflows across multiple heterogeneous resources distributed in the wide area, while at the same time shielding the user from the Grid detail [6].

Pegasus introduces the concepts of *abstract workflows* and *concrete workflows*. Abstract workflows are designed by domain scientists at the application or logical level, who specify input data, application components and their dependencies. They do not need to care about which physical resources to use for run-time execution; Concrete workflows at the execution level include not only the specific tasks to be executed but also the resources that would be used in the execution of the tasks [6].

Pegasus proposes to consider each application component as a service, in order to integrate Pegasus with the new *Open Services Grid Architecture* (OGSA) [54] that provides a syntactic description of the services. In addition, Pegasus proposes to develop ontologies of application components and data in their future work so that these ontologies can generate abstract workflows more flexibly from user requirements. Pegasus is also expected to employ ontologies to generate concrete workflows. Ontologies of Grid resources would allow the system to evaluate the suitability of given resources to provide a particular application service instance [52].

Discussion. From above systems, each of them uses a proprietary scientific workflow language, whose semantics has not yet been fully investigated and formalized. Second, each system has either no explicit architectural design or the architecture is proprietary and restricted greatly by the legacy system that the scientific workflow management system is built upon. For example, Kepler is built on the Ptolemy II system, and therefore, each new requirement that is needed by an SWFMS is based on extensions to the architecture of Ptolemy. Pegasus, on the other hand, is built upon Condor DAGMan [55] by adding another workflow mapper on the top of these two systems. Third, all these systems have different provenance models, not only in terms of what provenance information should be recorded, but also in terms of representation, storage, and querying models. We expect that the availability of a reference architecture can provide a basis for comparison between different systems and a guidance for the architectural design of an SWFMS in a specific scientific domain.

2.2 Workflow Integration

We survey the integration techniques applied in workflow systems in the aspect of services integration (Section 2.2.1) and data integration (Section 2.2.2).

2.2.1 Services Integration in Workflow Systems

The problem of integrating heterogeneous applications into workflows has been investigated by many business workflow systems [56, 57, 58, 59, 60, 61]. For example, the METEOR system [56] supports the programming of tasks by specifying possible heterogeneous task structures (e.g., transactional or non-transactional) and the interactions with external systems (e.g., DBMSs) or legacy applications. The Mobile prototype [57] introduces the concept of workflow application (WFA) which requires to encapsulate each application and provide a uniform interface to communicate with remote proxy objects for invocation. The METU-Flow system [58] provides a general interface for different task structures (e.g., transactional or non-transactional) and introduces a generic task object to each type of task structures.

However, these techniques are inapplicable to scientific workflows due to the fundamental differences between business workflows and scientific workflows: Business workflows tend to be controlflow oriented [62], while scientific workflows are often dataflow oriented [63, 3, 7, 4], resulting in different architectures and task models [64]. In particular, they are unable to abstract heterogeneous and distributed services and applications into uniform dataflow-based scientific workflow tasks (i.e., tasks with well-defined input and output ports).

Several scientific workflow management systems [3, 7, 4, 5, 6, 8, 65] have been developed over the past few years. Most of them allow users to program a task for invoking an external application or service, whose source codes are usually unavailable to use (e.g. Web services and command line applications). For example, the Triana system [8] requires programming tasks using Java to call a Windows command line application, while the VisTrails system [4] provides a Python package (i.e. *list2cmdline*) for users to program for similar purposes. Some systems provide built-in system-specific wrappers for the invocation of external services and applications. For example, the Kepler system [3] uses a Java class (e.g. *ExternalExecution*) for wrapping command line applications, and the Taverna system [7] uses a Java class (e.g. *LocalCommand*) for the similar purposes. Compared to the above systems, the Swift system [5] and the Pegasus system [6] mainly focus on the integration of Grid applications. Finally, even though the SODIUM system [65] provides a service model that abstracts three types of services (i.e. Web services, P2P services and Grid services) and employs plug-ins to support the invocation of these services, other applications and legacy systems need to be wrapped as one of these services first before they can be integrated into the system.

None of the above systems supports the flexible mappings between the input/output ports of the task interface and the inputs/outputs of the task component. In most cases, such mappings have to be performed by the development of custom workflow tasks that wrap the invocation of external service and applications and hardcode these mappings. This wrapper-programming approach is unnecessarily tedious, error-prone, and lacking the flexibility of supporting multiple task components and the dynamic binding capability between the task interface and the task component. Therefore, how to abstract heterogeneous and distributed services and applications into uniform workflow tasks remains a challenging problem. Our proposed task template model, task run model and their languages provide a programming language and platform independent framework to support such task abstraction and the execution of tasks constructed from heterogeneous and distributed services and applications.

2.2.2 Data Integration in Workflow Systems

During workflow design, third party autonomous services and applications are frequently used. Very often, these services and applications are syntactically mismatching or semantically incompatible, necessitating the use of a special kind of workflow components, called shims, to mediate them. A shim takes the output data of an upstream workflow task, performs some transformation, and then feeds the data to the input of a downstream task. The shimming problem has been widely recognized as an important problem in the community [15, 16], leading to much efforts in the development of shims [17], shim-aware workflow composition [15] and the suggestion of a new discipline called *shimology* [16].

The term "shims" and the shimming problem were first introduced in [17]. In an open world such as the Web, the shimming problem is unavoidable when third-party autonomous and heterogeneous services and applications are used to compose scientific workflows, but the output of one task is *incompatible* with the input of another task. Incompatibility comes in two forms: 1) Although the output of a task is syntactically compatible with the input of another task, they are still not compatible semantically. For example, both tasks might use `xsd:string` to encode underlying different complex data types. 2) Although the output of a task is syntactically incompatible with the input of another task, they could still be semantically equivalent. For example, DNA sequences might be represented in different formats and data types, which are semantically equivalent. In both cases, shims are proposed as the treatment of the shimming problem and are defined as the "software that transforms between closely related (either syntactically or semantically) in order to join outputs and inputs of two components" in [17].

Several shimming techniques have been proposed to address the shimming problem. Szomszor et al. [66] proposed an architecture to support the automatic translation between two semantically equivalent but syntactically different XML documents. This technique does not address the first form of incompatibility and the translation of other data types. Bowers and Ludäscher [16] proposed an ontology-based approach to the shimming problem by associating each port of a task with an XML-based *structural type* and an ontology-based *semantic type*, respectively. An output port of an upstream task can be directly connected to an input port of a downstream task if and only if these two ports are both semantically and syntactically compatible (called *semantically valid* and *structurally valid* in their terms). If two ports are semantically compatible but syntactically incompatible, then an XML shim is created whenever possible to mediate the two ports. Similarly, the solution is limited to shims that perform data transformation on XML data. Ambite and Kapoor [15] proposed a planning approach to automatically construct scientific workflows that process relational data. Shims can be automatically inserted into the workflow when necessary. However, only shims that process relational data are supported within this framework. Hull et al. [67] proposed that semantic types should be related to each other by other relationship types, such as *hasPart* in addition to the subsumption relationship. In this way, a richer types of shims can be created, such as *extractor* shims. A preliminary classification of shims are available in [17]. Existing scientific workflow management systems [3, 7, 4] provide limited support to the TYPE-I shimming problem; shims are visible in these systems.

Existing shimming techniques have two serious limitations. First, they produce scientific workflows that are cluttered with many visible shims. For example, a recent study of the 560 scientific workflows available from myExperiment (www.myexperiment.org) shows that over 30% of workflow tasks are shims. Ideally, these shims should be hidden from scientists so that they can better focus on functional components of workflows. Second, these techniques do not address TYPE-II shimming problem and thus require a user to write custom wrapper shim code around a task component according to the task programming model of a system. Moreover, these hard-coded implicit shims are *irreusable* across other tasks. Addressing TYPE-II shimming problem is more challenging due to the heterogeneity of task components and the needed flexible mapping between task ports and inputs/outputs of task components.

2.3 Workflow Scheduling

Workflow scheduling is one of the key problems in workflow management. It is a process that maps workflow tasks and their associated data to suitable resources and ordering the executions of these tasks, so that the workflow execution can be completed with the satisfaction of predefined objective functions. Generally, the scheduling problem is known as an NP-complete problem [18, 19], thus no known algorithms are able to generate the optimal solution within polynomial time. Therefore, many of heuristics based algorithms have been proposed in the literature. These algorithms can be classified into the heuristic and guided random search scheduling algorithms.

2.3.1 Heuristic Scheduling Algorithms

Various heuristic workflow scheduling algorithms are proposed to address different scheduling problems. The existing algorithms include independent task scheduling, clustering based scheduling, duplication based scheduling and list scheduling algorithms.

Independent task scheduling algorithms [68, 69, 70, 71, 72] schedule a collection of independent tasks with no data dependencies. For example, the UDA (User-Directed Assignment) [68, 69] algorithm maps each task of a workflow, in an arbitrary order, to the resource with the shortest execution time, without considering the available time of each resource. The Myopic algorithm [70], implemented in Condor DAGMan [55], retrieves a task from a set of an unmapped tasks in an arbitrary order, and then maps the task to the resource that is expected to complete it at the earliest time, until all tasks have been mapped. The Min-Min algorithm [71] iteratively selects a set of independent tasks and calculates the minimum estimated completion time for each task on all available resources. The task having the minimum estimated completion time is selected to be mapped to the best resource which is expected to complete at the earliest time. The intuition behind the algorithm is that mapping as many tasks as possible onto their best resources may result in a shorter makespan of the whole workflow. In contrast to the Min-Min algorithm, the Max-Min algorithm [71] selects tasks with the maximum estimated completion time within a set of tasks. Intuitively, the Max-Min algorithm attempts to minimize the delay caused by long-running tasks. Mapping long-running tasks onto the best resources at the first allows the parallel execution with the rest of short-running tasks. This algorithm may avoid the worst case in which all short-running tasks are executed first, and then the remaining long-running tasks are executed on several resources, while keeping the rest of resources idle. It has been shown that the Max-Min algorithm performs better than the Min-Min algorithm for some workflows which consist much more short-running tasks than long-running tasks [71, 72]. The Sufferage algorithm [71] sets priority to tasks based on their sufferge value, which is determined by the difference between its earliest completion time and its second earliest completion time. Sufferage is expected to perform better in case that there are dramatic performance differences between compute resources. Overall, independent task scheduling algorithms are easy to implement, but they are only suitable for simple workflow structures in which several tasks are required to be executed in sequential. Since these algorithms are proposed for independent tasks, so the data transmissions between different tasks are not applicable in this case.

The *clustering* based scheduling algorithms [73, 74, 75, 76, 77, 78, 79] can be applied to scheduling workflows onto unbounded number of resources [80]. These algorithms usually include two phases: at the first phase, tasks are partitioned into several clusters under the assumption that there are an unbounded number of resources. At the second phase, clusters are merged and scheduled on physical resources if the number of available resources is less than the cluster number. Tasks assigned in the same clusters are mapped onto one resource. For example, the Sarkar's algorithm [73] zeros the edge with the highest communication cost at the first phase if such an operation does not increase the parallel time of the workflow. Continue the next highest edge until all edges have been visited. At the second phase, clusters are scheduled to resources based on a priority list. The time complexity of the Sarkar's algorithm is $O(|D| \times (|T| + |D|))$, where $|T|$ is the number of tasks, $|D|$ is the number of inter-task data communications between a set of homogeneous processors. Yang and Gerasoulis [74, 75] claimed that zeroing the highest communication edges is not the best approach to reduce the parallel time. They introduced the DSC (dominant sequence clustering) algorithm [76] to compute the schedule and parallel time incrementally at the first phase, and then map clusters to physical processors and order the execution of tasks on each processor at the second phase. The total time complexity of the algorithm is $O((|T| + |D|) \log_{|T|})$. Since most of clustering scheduling algorithms are proposed for homogeneous multiprogramming environments where the physical machines are shared by multiple users and the number of available processors may not be known until run time, they are not applicable in heterogeneous computing environments where the execution time of each task and data transfer rates between tasks differ from one resource to another, and the number of resources can be requested on demand.

The *duplication* based scheduling algorithms [81, 82, 83, 84, 85, 86] use some resource idle time to duplicate tasks, which are also scheduled on other resources. For example, the TANH algorithm [81] traverses a workflow graph to compute critical information, such as earliest start and completion time, latest available start and completion time, and then clusters tasks based on such information. Then tasks are duplicated at idle time of resources and rearranged to decrease the overall execution time. Instead of duplicating tasks, Ranganathan et al. [85, 86] introduced dynamic replication strategies to improve data access. The performance is significantly improved when scheduling is performed according to data availability; however, replication are not able to be done instantaneously given the huge data size and bandwidth constraints. Although most of duplication based scheduling algorithms effectively decrease the large data transmission between tasks for complex structured workflows, they are not practical because of the significantly high time complexity. For example, the time complexity of the BTDH algorithm [82] and DSH algorithm [83] are in the order of $O(|T|^4)$; the time complexity of the CPFD algorithm [84] is $O(|D| \times |T|^2)$ for scheduling $|T|$ tasks.

The *list* scheduling algorithms [87, 88, 89, 90, 91, 92, 93, 94] prioritize each task of a workflow with a rank value and then order the execution of tasks according to their rank values; then tasks are selected in order of a priority list generated by the first phase and then are mapped to its optimal resource while minimizing a predefined cost function. For example, the MCP (modified critical path) algorithm [90] schedules a task that all its predecessors have completed execution onto an available resource that allows the task to start its execution at the earliest possible time. As MCP applies the earliest starting time principle, it may schedule all tasks to be executed onto one processor in the case that the communication cost is greater than the computation cost. Most of the above algorithms are mainly for homogeneous computing environments, while the HEFT (Heterogeneous Earliest-Finish-Time) algorithm [94] and the CPOP (Critical-Path-on-a-Processor) algorithm [94] provide practical solutions to schedule tasks on heterogeneous and distributed resources. The HEFT algorithm orders the upward rank value of tasks using the mean value of the task execution time and communication time over all heterogeneous resources. Then each selected task is assigned to the processor which minimizes its earliest finish time with an insertion-based approach. HEFT has an $O(|D| \times |R|)$ time complexity for $|R|$ processors. For a dense graph when the number of edges is proportional to $O(|T|^2)$, the time complexity is on the order of $O(|T|^2 \times |R|)$ [94]. The CPOP algorithm [94] prioritizes tasks using the sum of upward and downward rank values, and schedules tasks in critical path of the workflow graph onto resources that minimize the total execution time of these tasks. The time complexity of the CPOP algorithm is equal to $O(|T|^2 \times |R|)$. It has been shown in the literature that HEFT and CPOP significantly outperform other algorithms, such as the DLS (Dynamic Level Scheduling) algorithm [88], the MH (Mapping Heuristic) algorithm [95] and the LMT (Levelized Min Time) algorithm [96], in term of performance and cost metrics. Overall, list algorithms can generate good quality of scheduling results while keeping lower scheduling overhead; however, most of the list algorithms including the HEFT and CPOP algorithms were organically proposed for a bounded number of multiprocessor environments, they cannot directly applied to a services computing environment where the number of resources are provisioned by service requests and can be dynamically changed at run-time.

2.3.2 Guided Random Search Scheduling Algorithms

Guided Random Search based scheduling algorithms [97, 98, 99, 100, 101] provide general heuristics for solving the scheduling problem, which are usually applied to large-scale workflows. For example, a Greedy Randomized Adaptive Search Procedure (GRASP) [102] is an iterative randomized search technique, in which a number of iterations are performed to search a possible optimal solution for mapping tasks onto compute resources. Similar to

GRASP, genetic algorithms [103, 104] are of the most widely studied guided random search techniques in metaheuristics algorithms. The algorithms provide robust search techniques that allow a high-quality solution to be derived from a large search space in polynomial time by applying the principle of evolution. A genetic algorithm maintains a population of individuals that evolves over generations. The quality of an individual in the population is determined by a fitness function. The fitness value indicates how good the individual solution is compared to others in the population. Wang [97] encoded each chromosome with two separate parts: the matching string and the scheduling string. Matching string represents the assignment of tasks on machines while scheduling string represents the execution order of the tasks. Using genetic algorithms to schedule workflows in homogeneous and dedicated multiprocess system have been also proposed in [98, 99]. Although metaheuristics algorithms can produce optimized scheduling solution based on the performance of entire workflow and available resources, their execution times are significantly higher than other algorithms. It has been shown that the improvement of the GA-based solution to the second best solution was no more than 10 percent and the GA-based approach required around a minute to produce a solution while the other heuristics required an execution of a few seconds [105].

However, most of the above algorithms address the problem of assigning a workflow to a bounded number of resources. Even though some algorithms support unbounded number of resources for a workflow, they do not support dynamically changing the number of resources from the environment at runtime. Therefore, these algorithms are not applicable in services computing environments, in which the number of resources requested for a workflow can be elastically scaled out or scaled in on demand of the size of a workflow, thus motivates our research in this direction.

CHAPTER 3: REFERENCE ARCHITECTURE FOR SCIENTIFIC WORKFLOW MANAGEMENT SYSTEMS

As a software architecture for the design of high-level organization of computational elements and interactions between those elements is critical for any large software system [106], one of the fundamental issue missing in scientific workflow research is a proper foundation that can be adopted for SWFMS development. Although the reference architecture proposed by the Workflow Management Coalition has been well adopted in the development of different BWFMSs, existing architectures for BWFMSs are not appropriate for SWFMSs since business workflows are typically controlflow oriented, while scientific workflows tend to be dataflow oriented, introducing a new set of requirements and challenges for system development, which are described in Section 3.1. In response to these new requirements, we propose the first reference architecture for scientific workflow systems in Section 3.2, followed by the evaluations of five representative systems using the proposed reference architecture in Section 3.3.

3.1 Seven Key Architectural Requirements

In addition to the general requirements of scalability, reliability, extensibility, availability, and security, what are the key architectural requirements for an SWFMS? Based on a comprehensive study of the workflow literature from an architectural perspective [107] and our own experience from the development of the VIEW system, we identify the following seven key architectural requirements for an SWFMS:

R1: User interface customizability and user interaction support. In scientific workflows, scientists are often the end users to design, modify, run, re-run, and monitor scientific workflows. User friendly graphical user interfaces are critical to increase the usability of an SWFMS. Domain specific visualization capability is often needed to support the visualization of various workflow artifacts. The goal is to speed up the *exploratory process* of arriving at a proper workflow design with appropriate parameter values and input datasets that lead to sought-after scientific results. Therefore, a key architectural requirement is the flexibility of customizing the user interface according to different science and engineering disciplines, scientific domains or problems, or to an individual scientist's style, while reusing the same underlying workflow management framework. Customizing user interface should be localized and should not affect any other functional components of the system.

R2: Reproducibility support. Reproducibility is the fundamental principle of any science method. Scientific results produced from the execution of scientific workflows must be reproducible. Therefore, sufficient provenance information, including the derivation history of a data product, needs to be maintained in order to answer the following questions: What workflows or workflow steps are executed to produce this result? which versions of softwares and OSs are used? What parameter values are used? What input datasets have contributed to this result? What scientists' interactions are involved in producing this result? With such information, a scientific result can be reproduced in the same system or in other peer systems when necessary. Therefore, a key functional component for an SWFMS is the management of provenance metadata, from collection, representation, storage, querying, to visualization. Such a component is usually not required for a BWFMS.

R3: Heterogeneous and distributed services and software tools integration. Scientists often need to integrate and orchestrate a wide range of heterogeneous analytical and computational services and software tools into a scientific workflow for solving a complex scientific problem. Such services and software tools are usually written in various programming languages, invoked by different invocation mechanisms, and run in heterogeneous and distributed computing environments. Therefore, a key architectural requirement is to provide an abstraction of various services and software tools as *workflow tasks* (abr. task in this dissertation). Tasks not only keep scientists transparent to the heterogeneity and distribution of underlying task components, but also promote SWFMS extensibility so that the integration of future services and software tools whose interfaces and communication protocols are yet unknown does not affect other functional components of an SWFMS.

R4: Heterogeneous and distributed data product management. The execution of scientific workflows often consume and produce huge amounts of distributed data objects. These data objects can be of primitive or complex types, files in different sizes and formats, database tables, or data objects in other forms. Scientists are often overwhelmed and lost in the sea of heterogeneous and distributed data objects. Therefore, a key architectural requirement for an SWFMS is to provide an abstraction of these data objects as *data products*. Data products for SWFMSs include: 1) *workflow source data* that are registered into an SWFMS from external sources (produced by other systems, instruments, or experiments); 2) *workflow parameters* that are specified and tuned by users for each workflow run; 3) *workflow results* which consist of workflow intermediate and final results produced by workflow runs. Therefore, an SWFMS needs to support the efficient management of data products, including data product storage, archival, browsing, querying, access, movement, and visualization.

R5: High-end computing support. Today, many scientific problems need the support of high-end computing, such as Grid computing and Cloud computing [108]. Given the fast advance of high-end computing technology, a key architectural requirement for an SWFMS is to separate the science-focused and technology-independent problem solving environment from the underlying often fast advanced high-end computing infrastructure. In this way, domain scientists can focus on their science while utilizing the state-of-the-art computing technologies in a transparent fashion.

R6: Workflow monitoring and failure handling. The monitoring of the progress of the workflow execution is very important, particularly for long-running scientific workflows. Moreover, since scientific workflows are often designed and modified by scientists in an ad hoc fashion and can involve various distributed tasks that are accessed over network communications, many exceptions or failures can occur in an unforeseeable way. Finally, the complexity and scale of data analysis and computation in scientific workflows impose additional challenges on workflow monitoring and failure handling. Therefore, a key architectural requirement for an SWFMS is to provide the support for status and failure monitoring at various levels and the mechanism for catching, localizing, and handling failures automatically or with minimal human intervention.

R7: Interoperability. As more and more scientific research projects become collaborative in nature and involve multiple geographically distributed organizations, many scientific workflows are distributed and collaborative, consisting of multiple subworkflows, each of which might be managed by a different SWFMS. Therefore, a key architectural requirement for SWFMSs is to promote and facilitate the interoperability between different SWFMSs so that one SWFMS can take advantage of the software tool libraries and salient features provided by another SWFMS. The interoperability for SWFMSs lies in three levels: 1) *task-level interoperability*, which requires that various tasks and data products from different SWFMSs can interoperate one with another; 2) *workflow-level interoperability*, which requires that a scientific workflow in one SWFMS can be executed in or invoked by another SWFMS; and 3) *subsystem-level interoperability*, which requires that a subsystem in one SWFMS can be reused by or shared by different SWFMSs.

3.2 Proposed Reference Architecture

Although several SWFMSs have been developed over the past few years, their architectures are mostly system and domain specific and fail to satisfy some of the key architectural requirements for SWFMSs identified in Section 3.1. In this section, we propose a reference architecture for SWFMSs. As shown in Figure 3.1 (right), the reference architecture consists of four logical layers, seven major functional subsystems, and six interfaces. Figure 3.1 (left) shows a typical software stack of a scientific workflow application: on top of an operating system, a data management system and a service management is used by an SWFMS for data management and service management, respectively. A *scientific workflow application system* (SWFAS) is developed over an SWFMS by the introduction of additional domain-specific application data and functionalities.

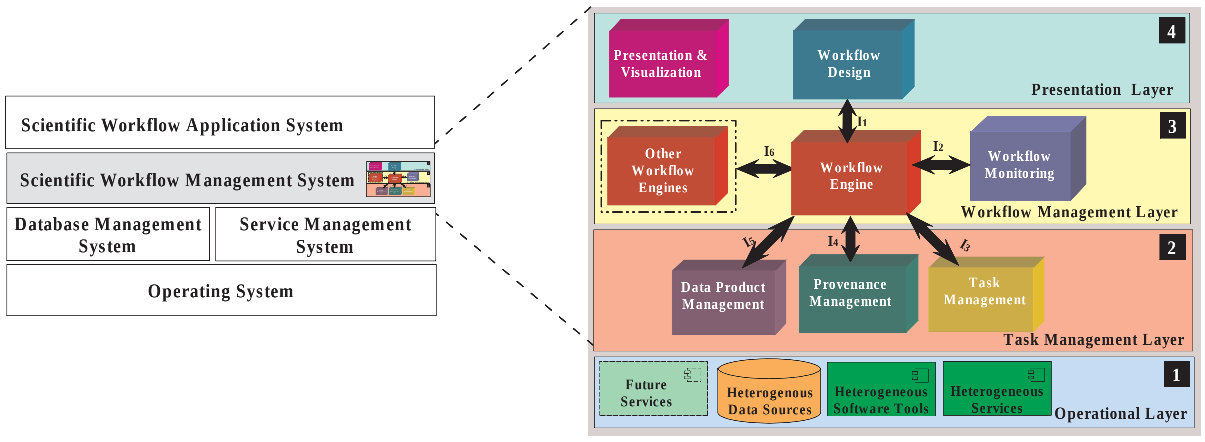


Figure 3.1: The position of an SWFMS within a software stack (left) and zoom-in view of the reference architecture for SWFMSs (right).

3.2.1 Layers

The first layer is the *Operational* Layer, which consists of a wide range of heterogeneous and distributed data sources, software tools, services, and their operational environments, including high-end computing environments. The separation of the Operational Layer from other layers isolates data sources, software tools, services, and their associated high-end computing environments from the scope of an SWFMS, thus satisfying requirement R5.

The second layer is called the *Task Management* Layer. Tasks are the building blocks of scientific workflows. Tasks consume input data products and produce output data products. At the same time, provenance is captured automatically to record the derivation history of a data product, including original data sources, intermediate data products, and the steps that are applied to produce the data product. This layer abstracts underlying heterogeneous data into data products, services and software tools into tasks, and provides efficient management for data products, tasks, and provenance metadata. Therefore, the Task Management Layer satisfies requirements R2, R3 and R4. Moreover, the separation of the Task Management Layer from the Operational Layer promotes the extensibility of the Operational Layer with new services and new high-end computing facilities, and localizes system evolution due to hardware or software advances to the interface between the Operational Layer and the Task Management Layer. The task-level interoperability requirement (R7: level 1) should be addressed in this layer.

The third layer is the *Workflow Management* Layer, which is responsible for the execution and monitoring of scientific workflows. At this layer, the building blocks of a scientific workflow are the tasks provided by the underlying Task Management Layer. In this layer, an execution of a scientific workflow is called a *workflow run*, which consists of an coordinated execution of tasks, each of which is called a *task run*. Therefore, the Workflow Management Layer addresses requirements R6 and R7. The separation of the Workflow Management Layer from the Task Management Layer concerns two aspects as follows: 1) it isolates the choice of a workflow model from the choice of a task model, so changes to the workflow structure do not need to affect the structures of tasks; and 2) it separates workflow scheduling from task execution, thus improves the performance and scalability of the whole system. The interoperability of workflows (requirement R7: level 2) has to be addressed by standardizing *workflow models*, *workflow run models* and *workflow languages*.

The fourth layer is the *Presentation* Layer, which provides the functionality of workflow design and various user interfaces and visualizations for all assets of the whole system. The *Presentation Layer* has interfaces to each lower layer (not shown in the figure for simplicity). The separation of the Presentation Layer from other layers provides the flexibility of customizing the user interfaces of the system and promotes the reusability of the rest of system components for different scientific domains. Thus, this separation supports requirement R1. The interoperability of workflows (requirement R7: level 2) should be addressed by standardizing the workflow layout (e.g. look-and-feel) at this layer.

3.2.2 Subsystems

The seven major functional subsystems correspond to the key functionalities required for an SWFMS. Although the reference architecture allows the introduction of additional subsystems and their features in each layer, this dissertation only focuses on the major subsystems and their essential functionalities.

The *Workflow Design* subsystem is responsible for the design and modification of scientific workflows. Workflow Design produces workflow specifications represented in a workflow specification language that supports a particular workflow model. One can design and modify a scientific workflow using a standalone or web-based workflow designer, which supports both graphical and scripting based design interfaces. The interoperability of workflows (requirement R7: level 2) should be addressed in this subsystem by the standardization of scientific workflow languages.

The *Presentation and Visualization* subsystem is very important especially for data-intensive and visualization-intensive scientific workflows, in which the presentation of workflows and visualization of various data products and provenance metadata in multi-dimensions are the key to gain insights and knowledge from large amount of data and metadata. These two subsystems are located at the *Presentation Layer* to meet requirement R1. In this subsystem, the interoperability of workflows (requirement R7: level 2) should be addressed by the standardization of scientific workflow layout.

The *Workflow Engine* subsystem is at the heart of the whole system and is the subsystem that provides management and execution environments for workflow runs. The Workflow Engine creates and executes workflow runs according to a workflow run model, which defines the state transitions of each scientific workflow and its constituent task runs. The interoperability of workflows (requirement R7: level 2) should be addressed by the standardization of interfaces, workflow models, and workflow run models, so that a scientific workflow or its constituent sub-workflows can be scheduled and executed in multiple workflow engines that are provided by various vendors. In SWFMSs, multiple workflow engine subsystems can be distributed, and each workflow engine can execute several workflows in parallel.

The *Workflow Monitoring* subsystem meets requirement R6 and is in charge of monitoring the status of workflow execution during workflow runtime and if failures occur, provides tools for failure handling [109].

The *Task Management* subsystem addresses heterogeneity and distribution issues (requirement R3) and provides management and execution environment for tasks, according to a *task model* and *task run model*, respectively. The interoperability of tasks between various workflow environments (requirement R7: level 1) can be addressed in this subsystem.

The *Provenance Management* subsystem meets requirement R2 and is mainly responsible for the management of scientific workflow provenance metadata, including their representation, storage, archival, searching, and visualization.

The *Data Product Management* subsystem meets requirement R4 and is mainly responsible for the management of heterogeneous data products. One key challenge for data product management is the heterogeneous and potentially distributed nature of data products, making efficient access and movement of data products an important research problem. The interoperability of data products between various workflow environments (requirement R7: level 1) can be addressed in this subsystem.

3.2.3 Interfaces

Each subsystem interacts with other subsystems by its interfaces. The interoperability between subsystems (requirement R7: level 3) in various SWFMSs should be addressed by standardizing the interfaces provided by each subsystem. In the reference architecture, six interfaces are explicitly defined, which show how the Workflow Engine interacts with other subsystems. The details of the interfaces between subsystems at the same layer are not shown in the figure for simplicity.

Interface I_1 provides a set of interfaces for the communications between Workflow Design subsystem and the Workflow Engine, so workflow specifications created by workflow design tools can be interpreted in the workflow execution environment. *Interface* I_2 provides a set of interfaces to report workflow run status from the Workflow Engine to the Workflow Monitor and to send back information from the Workflow Monitor to the Workflow Engine when dealing with exceptions, failure, and recovery. *Interface* I_3 provides a set of interfaces to deal with the communications between the Workflow Engine and the Task Management subsystem: the Workflow Engine subsystem sends requests to run each task, and the Task Management subsystem replies the task execution progress and acknowledges the Workflow Engine whether a task run completes or fails. *Interface* I_4 provides a set of interfaces for communication between the Workflow Engine and the Provenance Management for provenance tracking and reproducibility support. *Interface* I_5 provides a set of interfaces between the Workflow Engine and the Data Product Management subsystem: the Workflow Engine requests data product information from the Data Product Management subystem, and the Data Product Management subsystem responds to the request by acknowledging the availability of the required data product and delivering data or metadata as requested. Finally, *Interface* I_6 provides a set of interfaces to interoperate with other workflow engines. Workflow specifications can be passed through I_6 to another workflow engine for execution.

3.2.4 Summary

Due to the fundamental difference between scientific workflows and business workflows, our proposed reference architecture is significantly different from the reference architecture for BWFMSs. First, the reference architecture for SWFMSs contains the important components of provenance management and data product management to support scientific result reproducibility and to facilitate and speed up data analysis, respectively, which are not present in the reference architecture for BWFMSs. Second, the separation of the Presentation Layer from the Workflow Management Layer enables the support of user interaction and user interface customizability, thus reducing human cycles to scientific discovery. Third, the separation of the Workflow Management Layer from the Task Management Layer separates workflow engineering from task engineering, therefore allowing the parallel advancement of workflow management and task management. Finally, the separation of the Task Management Layer from the Operational Layer enables the separation of management of uniform workflow tasks from the heterogeneous low-level task implementation strategies and execution environments. Such a layered architectural design is important: for computer scientists, it enables abstractions and different independent implementations for each layer; for domain scientists, it provides the opportunity to develop a stable and familiar problem solving environment where rapid technologies can be leveraged but the details of which are shielded transparently from the scientists who need to focus on science itself.

3.3 System Evaluation Using the Reference Architecture

In this section, we evaluate five representative scientific workflow management systems using the proposed reference architecture: Taverna [110], Kepler [111], Triana [8], Pegasus [6], and Swift [5]. The analysis is performed based on the seven key architectural requirements in the context of the proposed reference architecture. Our evaluation criteria are as follows: if a system provides a full support to the specified requirement, a score of "+" will be assigned; if no support is provided to a particular requirement, a score of "-" will be assigned; a partial score of "+/-" will be assigned to a system when the support is clearly partial or when there is an ambiguity associated with such support. With respect to each requirement, we also describe a summary of the five systems to shed some lights on the state of the art. We have left out our own VIEW system in this study to avoid a biased evaluation.

The evaluation results are presented in Figure 3.2. We observe that Pegasus and Swift provide weak user interaction support (R1), mainly due to the technical challenge of supporting interaction in a batch-based grid system, while Taverna, Kepler, and Triana provide better user interaction support. Currently, almost all these systems have poor support to user interface customizability (R1) due to the tightly coupled nature between system interfaces and

Requirements	Taverna	Kepler	Triana	Pegasus	Swift
R1	+/-	+/-	+/-	-	-
R2	+	+	+	+	+
R3	+/-	+/-	+/-	-	-
R4	-	+/-	-	-	+/-
R5	+/-	+/-	+/-	+	+
R6	+/-	+/-	+/-	+/-	+/-
R7	-	-	-	-	-

Figure 3.2: Architectural evaluation of five scientific workflow management systems.

runtime subsystems. All the five systems currently support provenance (R2), emphasizing the importance of provenance in scientific workflow management systems. However, since the provenance module is closely coupled with its owner SWFMS in these systems, reuse of the provenance subsystem across other SWFMSs is difficult. Taverna, Kepler, and Triana have partial support to the integration of heterogeneous services and software tools (R3), while Pegasus and Swift only focus on grid-based applications. Therefore, a general framework that can provide an abstraction of heterogeneous services and applications as workflow tasks is still missing. Such a framework needs to clearly separate abstraction of a workflow task from its implementation and provides mapping and binding mechanisms between the inputs/outputs of a workflow task to the inputs/outputs to its wrapped service and application component. Although the importance of data management has been recently emphasized in the scientific workflow community [112], data product management (R4), particularly the abstraction of logical data products with transparent dataset representations, formats, and locations, is relatively an unexplored area in the scientific workflow community. For example, Pegasus and Swift mainly work at the level of files, while Taverna and Kepler work at the levels of XML messages, files, and database records. Only few systems provide some limited support to the abstraction: Kepler supports the notion of Nested Data Collections by using custom collection-oriented actors (co-actors), while Swift introduces the XDTM notation to define a mapping between the logical organization and the underlying physical structure of datasets, which are limited to files and directories so far. Currently, Pegasus and Swift have better support to high-end computing (R5); in the meanwhile, other systems are being enhanced in such support: Kepler and Taverna provide custom tasks to communicate with the Grid environment, while Triana uses the GAT interface to access Grid jobs. One challenge for data management is how to avoid the movement of large amounts of data back and forth from a workflow engine to the Grid environment, while seamlessly integrating workflow tasks that are services-based and Grid-based applications. All these five SWFMSs currently provide some degree of support to workflow monitoring and failure handling (R6), however, failure handling for large-scale and distributed scientific workflows remains a challenge. Finally, interoperability (R7) is poorly supported in all these SWFMSs although some limited pair-wise interoperability has been investigated. A community-based initiative such as the Open Provenance Model [113] is a good effort towards this direction, and we expect interoperability will become more important when more and more scientific projects become collaborative and need the integration of multiple SWFMSs.

CHAPTER 4: WORKFLOW INTEGRATION IN SCIENTIFIC WORKFLOWS

Today, scientific services and applications are developed by various research organizations originally as software tools for their own scientific problems and then are publicized for reusing them in solving other scientific problems. Therefore, it is very common that these software tools are written in various programming languages, invoked via different invocation mechanisms, and run in disparate computing environments. How to integrate these heterogeneous services and applications and execute them in a distributed environment is an open research problem. To address this problem, we firstly propose a task model and its supporting languages in Section 4.1. Then we propose our solution to solving shimming problems using our proposed task template language in Section 4.2. In Section 4.3, we introduce the proposed task run description language to enable distributed execution in heterogeneous environments. Finally, in Section 4.4 and Section 4.5, we describe our implementation of an SOA based task management in VIEW Task Manager and present a case study to validate the proposed models, languages and architecture.

4.1 Task Model

Tasks are the basic building blocks of a scientific workflow. A *task model* provides the modeling primitives to model design-time and run-time behaviors of workflow tasks. As shown in Figure 4.1, the design-time behavior of a task is modeled in a *task template model* and specified in a *task specification language* (TSL) as a *task template specification* (TTS), which often defines the interface of a task, and optionally, its implementation details. A set of task templates in a system constitute a *task library*, from which one can instantiate *task instances* for the creation of a scientific workflow.

During run-time, the execution status including run-time state and behavior of each task instance is maintained by a *task run*, which is modeled according to a *task run model* and

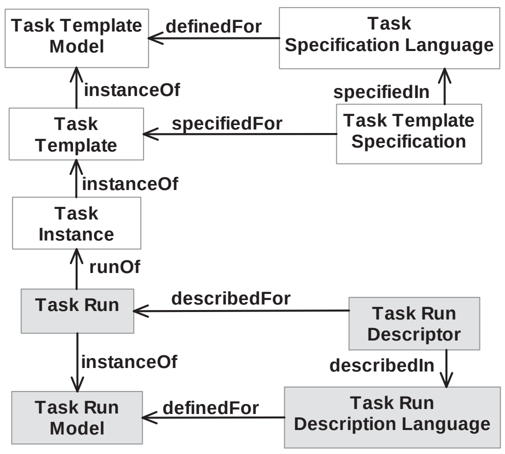


Figure 4.1: Main concepts and their relationships in a task model.

described in a *task run description language* as a *task run descriptor*. The task template model and the task run model are described in Section 4.1.1 and Section 4.1.2.

4.1.1 Task Template Model

In this section, we propose a task template model and its task specification language, TSL, for the specification of dataflow-based task templates, enabling the abstraction of various heterogeneous and distributed services and applications into uniform workflow tasks. Our proposed task template model is illustrated in Figure 4.2, consisting of the following three layers:

- *The logical layer* contains the *task interface* that models the input ports and output ports of a task template. In a scientific workflow, tasks are connected to one another via these ports through data channels. During workflow execution, tasks communicate with each other by passing data through data channels. The data type of each port is also defined as part of the task interface.

- *The physical layer* contains one or more *task components* that model the services or/and applications that are used to implement the task. The heterogeneous characteristics of a task component is modeled in this layer, including task type, inputs, outputs, location, invocation mechanism, authentication and protocol if needed.

- *The mapping layer* essentially consists of a list of mapping instructions that perform the mapping between the input/output ports of the task interface and the inputs/outputs of the task component. For each mapping, a shim is incorporated only if the type of input/output port and input/output are incompatible. All shims between input ports and inputs are formed an *inputports-to-inputs shim set*; while all shims between outputs and output ports are formed an *outputs-to-outputports shim set*.

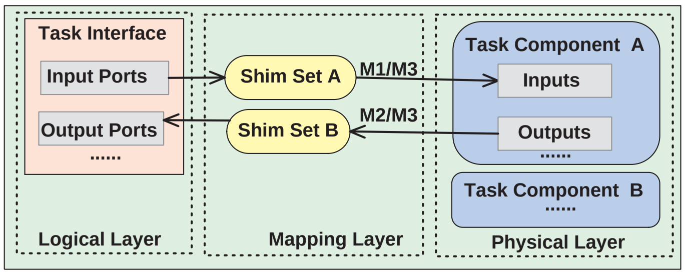


Figure 4.2: An extensible task template model.

The separation of the logical layer from the physical layer not only hides the implementation details of a task from its interface, thus providing a uniform interface of a task to the workflow engine, but also brings the opportunity to integrate various heterogeneous and distributed services and applications into a scientific workflow in a uniform way. However, the integration of heterogeneous services and applications into scientific workflows is challenging since these services/applications are often written in various programming languages, invoked via different invocation mechanisms and run in disparate computing environments. Currently, our proposed task template model focuses on the modeling of the following aspects of the heterogeneity of a task component:

- *Heterogeneous inputs of a task component.* A task component can take inputs from command line arguments (user-specified or constant), environment variables, input files, communication messages (e.g., SOAP messages for Web services), and the system standard input, etc.

- *Heterogeneous outputs of a task component.* A task component can produce outputs as environment variables, files, communication messages, the system standard output, the exit code, and the standard error, etc.

- *Heterogeneous invocation mechanisms.* Based on different computing environments, the types and locations of executables, various local and remote invocation mechanisms are modeled.

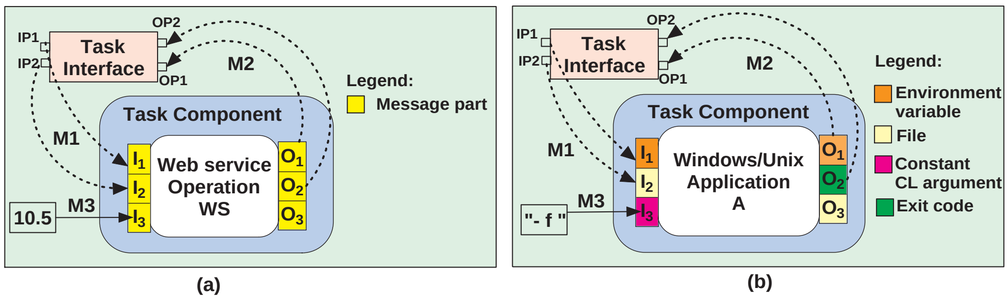


Figure 4.3: (a) - (b) static mappings between input/output ports of a task interface and inputs/outputs of the task components WS and A.

To hide the heterogeneous characteristics of a task component from the task interface, all the above heterogeneous aspects of a task component are modeled in the physical layer, while the mapping layer models the following three kinds of mappings between the input/output ports of the task interface and the heterogeneous inputs/outputs of a task component:

- The *inputports-to-inputs mapping* (M1) specifies how the input data taken from an input port IP_i of a task is mapped to an input I_j of the task component C. If IP_i is not mapped, then any data from IP_i will not be used by C. For each shim S in an inputports-to-inputs shim set, M1 contains the mapping between IP_i and the input of S and the mapping between output of S and I_j.

- The *outputs-to-outputports mapping* (M2) specifies how the output data produced from an output O_i of a task component is mapped back to an output port OP_j of the task.

Similarly, if an output of a task component is not mapped, then such output data is discarded. For each shim S in an outputs-to-outputports shim set, M2 contains the mapping between O_i and the input of S and the mapping between output of S and OP_j.

- The *constant mapping* (M3) specifies a constant that will be assigned to an input of the task component before the execution of the task component. A constant mapping can also be used to assign a constant value to an output port of a task when the execution of the task component completes. Such flexibility is important to improve the configurability of a task template.

Figures 4.3.(a) - (b) illustrate two cases of the application of our proposed task template model: Web services and Windows applications. For simplicity, shims are not shown in these mappings. For M1, in a Web service operation WS, as shown in Figure 4.2.(a), the input port IP_1 is mapped to I_1, one part of the request message of WS; the input port IP_2 is mapped to I_2, a second part of the request message. For M3, a constant 10.5 is assigned to I_3, a third part of the request message. For M2, a part of the response message O_1 is mapped to the output port OP_1; O_2, a second part of the response message, is mapped to the output port OP_2; and O_3, a third part of the response message is not mapped, indicating that its value is discarded and never used afterwards. For Windows/Unix applications, both mappings are more sophisticated due to the rich modes of inputs and outputs. As illustrated in Figure 4.2.(b), for M1, the input port IP_1 is mapped to environment variable I_1, requiring that this environment variable be assigned the value from IP_1 before the execution of a Windows/Unix Application A, and such value will be taken as A's input; the input port IP_2 is mapped to file I_2, indicating that a file I_2 needs to be created with the content from IP_2 before the execution of A. For M3, a constant string of “-f” is assigned to I_3, indicating that the invocation of A is achieved via a constant command line argument of “-f”. For M2, environment variable O_1 is mapped to output port OP_1, thus, after the execution of A, O_1 is produced as an environment variable and its value will be assigned to output port OP_1; the exit code O_2 is mapped to output port OP_2, therefore its value will be assigned to OP_2 after the execution of A; the execution of A will produce file O_3; however, since O_3 is not mapped, this file is discarded and will not be used afterwards. An optimization algorithm can delete such files to reclaim storage resources.

4.1.2 Task Run Model

A task run captures the state and run-time behaviors of the execution of a task instance. In this section, we propose a task run model and its task run description language, TRDL, for the description of task runs, enabling the execution of task instances constructed from heterogeneous services and applications. Our proposed task run model is illustrated in Figure 4.4, consisting of the following three layers:

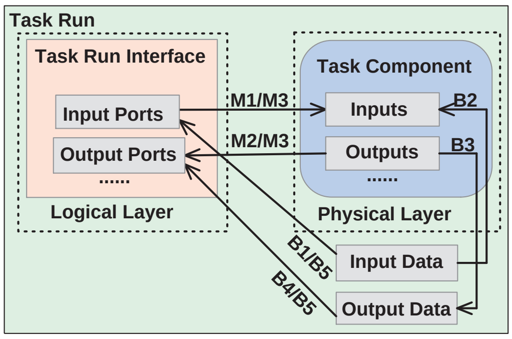


Figure 4.4: An extensible task run model.

- *The logical layer* corresponds to the logical layer in the task template model. It defines the input and output ports of a task interface, the state of the task run (one state of a state transition diagram), and additional run-time information, including the task run ID, task instance ID, workflow instance ID, and workflow run ID.

- *The physical layer* corresponds to the physical layer in the task template model. It contains one of task components that are contained in the physical layer of the task template model.

- *The binding layer* corresponds to the mapping layer in the task template model. Mappings are instructions for bindings, while bindings are the realization of mappings and involve the dynamic binding of data to the input/output ports of task interfaces and the inputs/outputs of task components.

One important functionality of a task run is the status tracking of the execution of a task instance. This is achieved by the update of the status field in the logical layer of the task run model, following a task run state transition diagram shown in Figure 4.5: A task run is created after the initiation of a task execution. Then, after an execution host is acquired (*task mapping*), the task run enters the *Mapped* state. Next, all the input data needed for the execution of the task are moved to the host where the task is mapped (*data movement*), and the task run enters into the *Ready* state. The task is then invoked (*task invocation*), which transits the task run into the *Executing* state. Finally, depending on if the task run terminates successfully or unsuccessfully, the task run enters either the *Success* state or the *Failed* state.

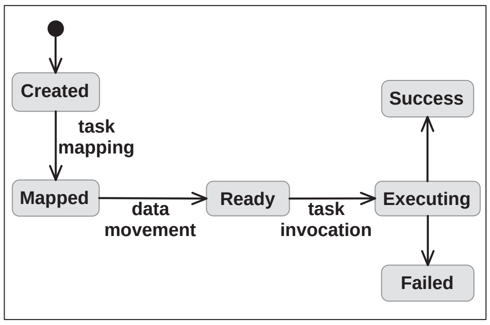


Figure 4.5: Task run state transition diagram.

The logical layer maintains the status of a task run at an abstract level, regardless of the implementation details of its task component, the separation of the logical layer from the physical layer provides the opportunity of dynamic binding between the task run interface and the task component that implements it. This is useful when there are multiple task components that implement the same task run interface; when the execution of one fails, another task component can be bound to carry out the same required functionality transparently from the workflow engine. Further more, dynamic data bindings for data, input/ouput ports of the task interface, and inputs/outputs of the task component can be performed at different stages of the lifecycle of a task run. We consider the following five types of bindings in our task run model:

- The *inputdata-to-inputports binding* (B1) describes that some input data are bound to the input ports of the task run interface. This binding is typically performed after a task run is first created.
- The *inputdata-to-inputs binding* (B2) describes that input data are bound to the inputs of a task component. This binding is derived from B1 and the inputports-to-inputs mapping (M1) specified in the task template. This binding is typically performed after a task is mapped to an appropriate execution host.
- The *outputs-to-outputdata binding* (B3) describes that the outputs of the task component are bound to the output data produced as the result of executing the task component. This binding is typically performed after the successful execution of the task component.
- The *outputdata-to-outputports binding* (B4) describes that the output data are bound to the output ports of the task run interface. This binding is derived from B3 and the outputs-to-outputports mapping (M2) specified in the task template. This binding is typically performed right after the outputs-to-outputdata binding (B3).
- The *constant binding* (B5) describes that a constant value is bound to an input/ouput of the task component before/after its execution. This binding is typically performed right before a task run enters the *Ready* state or after a task run enters the *Success* state.

Figures 4.6.(a) - (c) illustrate some snapshots of a task run constructed from a Windows application W_j at the states of *Created*, *Ready*, and *Success*, respectively. At the *Created* state, two data products, D_1 and D_2, are bound to input ports IP_1 and IP_2, respectively. At the *Ready* state, using the mapping information, D_1 is bound to I_1 since IP_1 is mapped to I_1, and

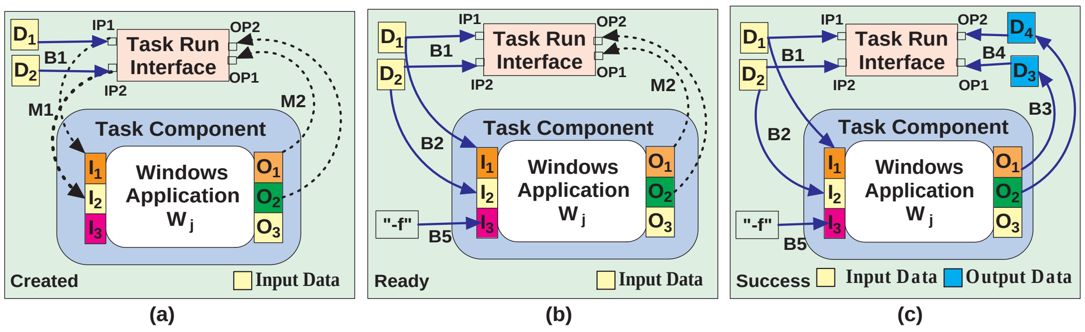


Figure 4.6: (a) - (c): dynamic bindings among input/output ports, inputs/outputs and input/output data at three states.

D_2 is bound to I_2 since IP_2 is mapped to I_2. Moreover, a constant “-f” is bound to I_3 using the constant mapping information (M3) for I_3. Finally, at the *Success* state, two data products, D_3 and D_4 are produced from outputs of the task component O_1 and O_2, respectively. Using the outputs-to-outputports mapping (M2) information, they are bound to output ports OP_1 and OP_2, respectively.

4.2 Addressing Shimming Problems in Scientific Workflows

We refer to the above shimming problem as *TYPE-I shimming problem*, which occurs at *the workflow level* due to the incompatibility of output ports of an upstream task with the input ports of a downstream task. For example, in Figure 4.7.(a), when the output port OP_2 of upstream task T_1 is incompatible with the input port IP_3 of downstream task T_2, a shim is needed to mediate them. While still not recognized by the community, we identify a second type of shimming problem, called *TYPE-II shimming problem* that occurs at the *task level* when tasks are created from third-party heterogeneous services and applications (called *task components*) and there is incompatibility between task ports and inputs/outputs of task components. For example, in Figure 4.7.(b), although $T_1.OP_i$ $(i = 1, 2, 3)$ and $T_2.IP_i$ $(j = 2, 3, 1)$ are compatible, inside T_2, input port IP_2 is incompatible with input I_2 of task component C and output O_3 of C is incompatible with output port OP_3 of task T_2.

Based on the task template model presented in Section 4.1.1, our proposed approach to TYPE-II and TYPE-I shimming problems are addressed in Section 4.2.1 and Section 4.2.2, followed by the summary of advantages of our approach in Section 4.2.3 and a case study in Section 4.2.4.

4.2.1 Addressing TYPE-II Shimming Problem Using TSL

According to the above task template model, an XML-based task template specification language, called *TSL*, is proposed to model heterogeneous and distributed services and applications, including shims. The XML schema of TSL is shown in Appendix A. In TSL, both shims and functional task components are uniformly modeled as *task components* with the *shim* role and the *functional* role, respectively. A task component can be registered with a system with one role or both roles.

Figure 4.8 presents an example of task template specification (TTS) for a task template written in TSL. The logical layer, the physical layer, and the mapping layer are realized by the `taskInterface` element, the `taskComponents` element and the `mappings` element, respectively. At the logical layer, the `taskInterface` element contains sub-elements `inputPorts` and `outputPorts` to define the input and output ports of the task template.

At the physical layer, the `taskComponents` element contains a set of `taskComponent` elements, modeling either functional task components (specified by `role = "functional"`)

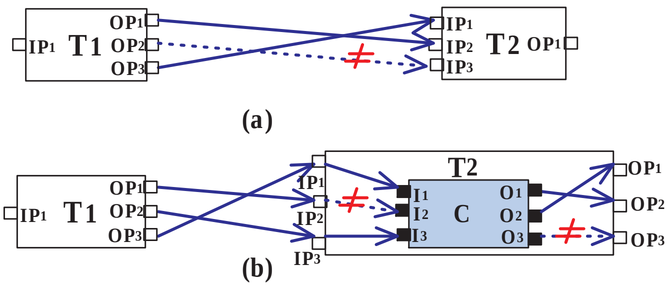


Figure 4.7: (a) The TYPE-I shimming problem; (b) The TYPE-II shimming problem. ($\neq$: mismatch)

```xml
<tsl:taskTemplate version="1.0" xmlns:tsl="http://view/tsl">
  <taskInterface id ="T67">
     <taskName>Mesh Hole Fill</taskName>
     <taskDescription>Fill holes in the iso-surface. </taskDescription>
     <inputPorts number="3">
        <port id ="IP87" default = "Yes">
           <portType>File(TET)</portType>
           <portDescription>An obj mesh format file of iso-surface.</portDescription>
           <portDefaultValue>...</portDefaultValue>
        </port>
        <port id ="IP88" default = "Yes">...</port>
        <port id ="IP89" default = "Yes">...</port>
      </inputPorts>
     <outputPorts number="2">
        <port id ="OP83">
            <portType>File(OBJ)</portType>
            <portDescription>An obj mesh format file with holes covered.</portDescription>
        </port>
        <port id ="OP84">... </port>
     </outputPorts>
  </taskInterface>
  <taskComponents>
     <taskComponent id ="TC101" default = "Yes" role="functional">
     <taskType>Windows Application </taskType>
     <executable>file://localhost/OBJ_FILL.exe</executable>
     <taskDescription> converting a TET file into an OBJ file. </taskDescription>
     <AppName>OBJ_FILL</AppName>
     <inputs>
        <input id = "I123" mode="FILE" fileName="/OBJFILL.obj" type="FILE(OBJ)"/>
        <input id = "I125" mode="EnviornmentVariable" envName="inputEnv" type="String"/>
        <input id = "I126" mode="ConstantCLArg" argName="inputCLArg" type="String"/>
    </inputs>
    <outputs>
        <output id = "O125" mode="FILE" fileName="Subj_hfobj" type="FILE(OBJ)"/>
        <output id = "O124" mode="ExitCode" name="ExitReturnValue" type="Integer"/>
     </outputs>
     <taskInvocation>
        <operatingSystem>Windows</operatingSystem>
        <invocationMode>Local</invocationMode>
        <interactionMode>No</interactionMode>
        <invocationAuthentication>...</invocationAuthentication>
      </taskInvocation>
    </taskComponent>
    <taskComponent  id ="TC103" default = "No" role="functional">
      <taskType>Web Service</taskType> ...
    </taskComponent>
     <taskComponent id ="TC102" default = "No" role="shim">
    <taskType>Windows Application </taskType>
    <taskDescription> converting a TET file into an OBJ file. </taskDescription>
     <executable>file://localhost/TET_FILL.exe</executable>
     <AppName>TET_FILL</AppName>
     <inputs>
        <input id = "I17" mode="FILE" fileName="/input.tet" type="FILE(TET)"/>
    </inputs>
    <outputs>
        <output id = "O13" mode="FILE" fileName="/output.obj" type="FILE(OBJ)"/>
     </outputs>
     <taskInvocation>
        <operatingSystem>Windows</operatingSystem>
        <invocationMode>Local</invocationMode>
        <interactionMode>No</interactionMode>
        <invocationAuthentication>...</invocationAuthentication>
      </taskInvocation>
    </taskComponent>
  </taskComponents>
  <mappings>
     <mapping id ="TC101">
        <inputmapping from="IP87" to="I123" shimming = "Yes"/>
            <shims  id = "TC102">
                <shimming from="IP87" to="I17">
                <shimming from="O13" to="I123">
           </shims>
       </inputmapping>
        <inputmapping from="IP88" to="I125" shimming = "No"/>
        <inputmapping from="IP89" to="I126" shimming = "No"/>
       <assign from="-f" to="IP89" />
        <outputmapping from="O125" to="OP83" />
        <outputmapping from="O124" to="OP84" />
     </mapping>
     <mapping id ="TC103"> ... </mapping>
  </mappings>
<taskInstances>
  <taskInstance id ="51">
     <taskComponent id ="TC101"/>
  </taskInstance>
  <taskInstance id ="52">
     <taskComponent id ="TC103"/>
  </taskInstance>
</taskInstances>
</tsl:taskTemplate>
```

Figure 4.8: An example of a task template specification.

or shim task components (specified by `role = "shims"`). Each functional `taskComponent` element specifies one possible implementation of the task interface of the task template. Similar to functional task components, shims are heterogeneous, distributed and system-independent. For each task component (shim or functional), we model its input/output information, invocation details, such as operating system, invocation mode (e.g., local or remote), interaction mode (interactive or non-interactive), and authentication information. Shims are introduced into `taskComponents` only if there is an inputports-to-inputs shim set or outputs-to-outputports shim set as a result of the TYPE-II shimming problem.

At the mapping layer, the `mappings` element contains the instructions for M1 (by the `inputmapping` element), M2 (by the `outputmapping` element) and M3 (by the `assign` element). If there is no shim for an inputmapping/outputmapping, the `shim` attribute inside the `inputmapping/outputmapping` is set to “No”; otherwise (`shim` = “Yes”), each `shimmings` element is encoded inside an `inputmapping` or `outputmapping` element. A `shimmings` element is uniquely identified by a shim’s `taskComponent id`. The `shimming` elements are encoded inside the `shimmings` element to provide the mappings among input/output ports, inputs/outputs of task components and input/outputs of shims.

The `taskInstances` element contains all task instances that are instantiated from the same task template and hence share the same task interface. In our model, we consider all functional task components in a task template is *functionally* equivalent but might have different implementations and deployments and thus might provide different types of inputs and outputs. Each task instance uses a unique functional component, which uniquely identifies the necessary mapping and shimming to provide the same task interface. Therefore, in TTS, each task instance encoded in the `taskInstance` element contains one specific functional task component from alternative task components provided by the task template. The taskComponent’s `id` inside each `taskInstance` can be used to retrieve the corresponding inputmapping and outputmapping of this task component.

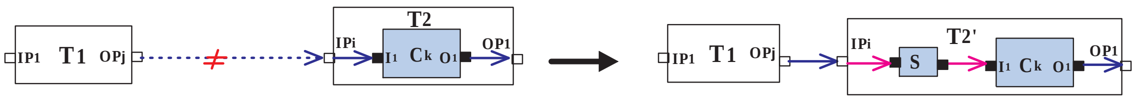


Figure 4.9: Reducing the TYPE-I shimming problem to the TYPE-II shimming problem.

Essentially, our example of task template specification, called *Mesh Hole Fill (MHF)*, provides three input ports and two output ports at the interface. MHF encapsulates two functional task components: one is called OBJ_FILL (`taskComponent id = TC101`), a Windows application that can be locally executed without user interaction. Another functional component encapsulated in MHF is developed as a Web service (`taskComponent id = TC103`). OBJ_FILL has three inputs with the modes of file, environment variable and constant command-line argument. Two outputs are defined with the modes of file and exit code. As the input of OBJ_FILL (`input id = I123`) is incompatible with the inputport (`port id=I123`) in input mapping, a shim (`taskComponent id = TC102`) is incorporated into the physical layer and the mapping layer of the TTS.

4.2.2 Addressing the TYPE-I Shimming Problem Using TSL

Next, we propose a reduction algorithm that reduces the TYPE-I shimming problem to the TYPE-II shimming problem and thus provide a transparent solution to both problems. As shown in Figure 4.9.(a), given two task instances T_1 and T_2, in which T_2 encapsulates functional task component C_k. When the type of output port $T_1.OP_j$ is incompatible with the type of input port $T_2.IP_i$, a TYPE-I shimming problem occurs. A new task template T_2' can be created from T_2's task template by encapsulating an appropriate shim S and C_k inside, and then an instance of T_2' can be used as a replacement of T_2. The pseudocode of the reduction algorithm, *ReduceTYPE-I2TYPE-II*, is sketched in Figure 4.10. First, the TTS of T_2' is copied from the TTS of T_2. Second, if possible, a suitable shim S is retrieved automatically based on the types of $T_1.OP_i$ and $T_2.IP_j$. Finally, different layers of T_2' are updated accordingly, in particular, T_2''s input port is mapped to S's input and S's output is mapped to the input of the task component C_k.

```
Algorithm: ReduceTYPE-I2TYPE-II
Input: TypeOf($T_1.OP_i$): a type of a task instance $T_1$'s output port $OP_i$, and
       TypeOf($T_2.IP_j$): a type of task instance $T_2$'s input port $IP_j$
Output: a new task instance $T_2'$ initialized by a new task template $T$
Begin
(1) If TYPE-I problem occurs
(2) Then Retrieve a shim from system or third-party
(3)       If $\exists$ a shim $S$ and TypeOf($S.in$) = TypeOf($T_1.OP_i$) and TypeOf($S.out$) = TypeOf($T_2.IP_j$)
(4)       Then
(5)            Create new task template $T$ by copying $T_2$'s TTS
(6)            Initialize a instance $T_2'$ based on $T$ TypeOf($T_2'.IP_j$) = TypeOf($T_1.OP_i$) /*update TTS's logical layer*/
(8)            Add $S$ into $T$'s taskComponents /*update TTS's physical layer*/
(9)            Map $T_2'.IP_j$ to $S.in$ /*update TTS's mapping layer*/
(10)           Map $S.out$ to the input of $T_2'$'s task component $C_k$
(11)      Else
(12)           Report to Type Match Error
(13)Else
(14) No shim required to reduce
End Algorithm
```

Figure 4.10: Algorithm ReduceTYPE-I2TYPE-II

4.2.3 Advantages of Our Approach

we identify the following advantages of our shimming approach:

1) *Transparent shimming*. This is the first shimming technique that hides all shimming and mapping details inside a task interface and thus produces scientific workflows in which all shims are invisible. As a result, a scientist can better focus on the functional part of a scientific workflow without being distracted by the clutter of shims, which are usually not science-relevant to the scientist but are technically needed.

2) *Addressing both TYPE-I and TYPE-II shimming problems*. This is the first solution that addresses the TYPE-II shimming problem. Moreover, our approach enables the reduction of the TYPE-I shimming problem to the TYPE-II shimming problem, providing a consistent solution to both types of shimming problems.

3) *System and language independent*. Since our shimming technique is based on an XML-based TSL language, which models all the details of abstraction, shimming and mapping. TSL can be implemented by different systems using different languages and thus provides a system and language independent solution.

4) *Reusable and extensible*. In our approach, similar to functional task components, shims can be arbitrary local and remote heterogeneous services and application written in various languages and run in different platforms. As a result, shims are reusable across tasks, workflows and systems. Moreover, TSL is easily extensible for more sophisticated shimming techniques, such as the composition of basic shims to construct composite shims.

4.2.4 Case Study: Shimming in Volumn Data Surface Extraction

Figure 4.11 presents a typical scientific workflow designed in VIEW 2.1 for surface extraction from volume data, a required preprocessing process for surface analysis. The workflow is composed of three task instances: the first is the *Iso-Surfacer* task instance which uses the marching cubes algorithm to extract the surface from volume data. The second task instance, called *TET_FILL*, analyzes the extracted surface to identify holes that are generated in an image file and fill them. The resulting surface is rendered in a 3D-interactive display using the *VTK_Display* task instance as shown in Figure 4.11.(c). The data types of input/output ports for each task instance are listed as follows: the Iso-surfacer task instance reads a volume file formatted as *VOL* from its inputport, and output a file formatted as *OBJ*; the inputport and outputport of TET_FILL task instance are typed as File (OBJ); The VTK_Display task instance read a VTK file and then visualize it on a display window.

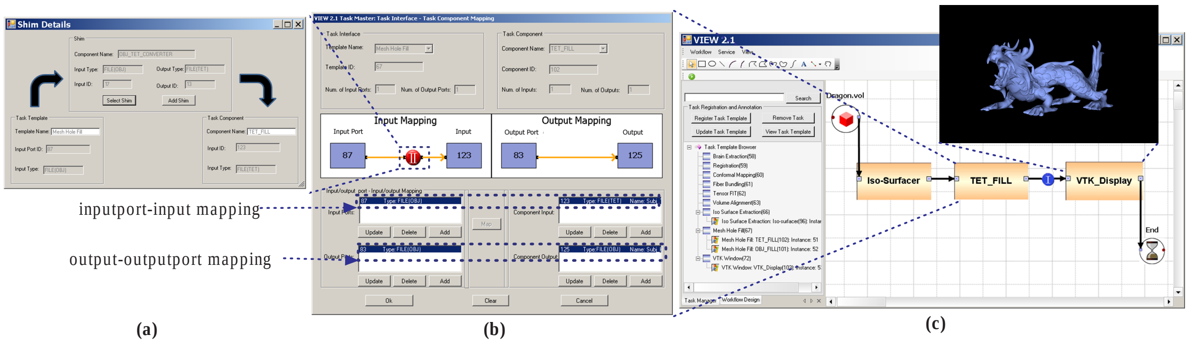


Figure 4.11: A scientific workflow composed in the VIEW 2.1 system with shims to the TYPE-I and TYPE-II problem.

The *TET_FILL* task instance is initialized by the *Mesh Fill Hole* task template which encapsulates two task components: the *TET_FILL* task component is a third-party Windows application using C++, invoked by a TYPE-D Executor. Another task component is called *OBJ_FILL*, implemented by a Web service that receives and outputs a datastream encoded in SOAP messages. This task component is invoked by a TYPE-B Executor. The *Mesh Fill Hole* task template's TSL can be viewed by Task Template Browser in Figure 4.11.(c) and stored in the VIEW Task Master.

Figure 4.11.(c) illustrates the Type-I shim that occurs between TET_FILL and VTK_Display task instances. The input port of VTK_Display is typed as File (VTK), incompatible with the type of TET_FILL's output defined as File (OBJ), then a Type-I shimming problem is detected automatically by the system (see the blue Type-I shimming detection icon in Figure 4.11.(c)). By clicking the icon, the system allows scientists to either select system-provided shims or register any third party shims if there is no existing shim available. In addition, the system allows scientists to automatically hide shims inside a task instance by applying our proposed ReducingType-I2Type-II algorithm.

The Type-II shim problem in this workflow occurs when mapping from the TET_FILL task instance's inputport to an input of its task component. The type of the input port is defined as File (OBJ), while the input requires a tetrahedral mesh file typed as File (TET). The incompatibility is automatically detected by system with the red Type-II shimming detection icon in Figure 4.11.(b). After clicking the icon, a system-provided shim called OBJ_TET_CONVERTER is automatically applied to the input mapping. Figure 4.11.(a) shows the shimming between the input port (ID:87,Type:File (OBJ)) and the shim's input (ID:17,Type:File (OBJ)), and the shimming between the shim's output (ID:13,Type:File (TET)) and the task component input (ID:123,Type: File (TET)). The implementation details of the OBJ_TET_CONVERTER shim is encoded in the Mesh Fill Hole task template's TTS, which is implemented as a Windows application using C++ and invoked remotely by a TYPE-A Executor.

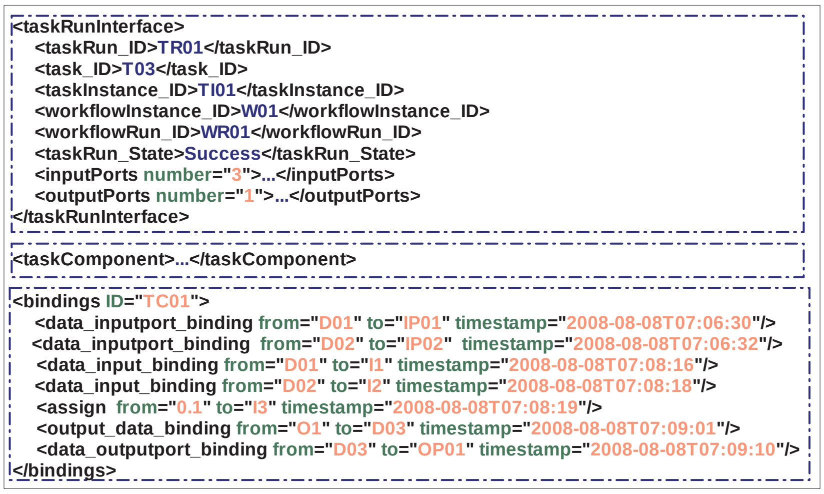
4.3 Addressing Heterogeneity Using TRDL

We have proposed an XML-based task run description language, called TRDL, to realize our proposed task run model. The XML schema of TRDL is shown in Appendix B. Figure 4.12 presents a snapshot of a task run descriptor at the *Success* state, written in TRDL. The logical layer, the physical layer, and the binding layer are realized by the `taskRunInterface` element, the `taskComponent` element and the `bindings` element, respectively. First, the `taskRunInterface` element contains subelements `inputPorts` and `outputPorts` to describe the input and output ports of the task, subelements `workflowInstance_ID` and `workflowRun_ID` to describe the workflow context that the task is executed within, and the *taskRun_State* element to describe the state of the task run. Second, the `taskComponent` element contains the description of the implementation details of the task interface; such information is obtained from the corresponding task template specification. Finally, the *bindings* element records all the bindings that have occurred up to the point of the state of the task run: the `data_inputport_binding` element records the inputdata-to-inputports binding (B1), the `data_input_binding` element records the inputdata-to-inputs binding (B2), the `output_to_data_binding` records the outputs-to-outputdata binding (B3), the `data_outputport_binding` records the outputdata-to-outputports binding (B4), and the `assign` element records the constant binding (B5). Each binding is associated with a timestamp attribute to record the time that the binding occurs.

4.4 Addressing Heterogeneity in VIEW Task Manager

As shown in Figure 4.13, the architecture of the Task Manager consists of a *Task Master* and a set of *Task Executors*. The Task Master manages all task templates, task instances, and task runs, while Task Executors are responsible for the invocation and execution of various heterogeneous task components. Four types of Task Executors are proposed but the extensibility are provided for future types of Task Executors:

1) A *TYPE-A* executor provides an execution environment mostly for user-interaction and



```
<taskRunInterface>
    <taskRun_ID>TR01</taskRun_ID>
    <task_ID>T03</task_ID>
    <taskInstance_ID>TI01</taskInstance_ID>
    <workflowInstance_ID>W01</workflowInstance_ID>
    <workflowRun_ID>WR01</workflowRun_ID>
    <taskRun_State>Success</taskRun_State>
    <inputPorts number="3">...</inputPorts>
    <outputPorts number="1">...</outputPorts>
</taskRunInterface>
<taskComponent>...</taskComponent>
<bindings ID="TC01">
    <data_inputport_binding from="D01" to="IP01" timestamp="2008-08-08T07:06:30"/>
    <data_inputport_binding  from="D02" to="IP02"  timestamp="2008-08-08T07:06:32"/>
    <data_input_binding from="D01" to="I1" timestamp="2008-08-08T07:08:16"/>
    <data_input_binding from="D02" to="I2" timestamp="2008-08-08T07:08:18"/>
    <assign  from="0.1" to="I3" timestamp="2008-08-08T07:08:19"/>
    <output_data_binding from="O1" to="D03" timestamp="2008-08-08T07:09:01"/>
    <data_outputport_binding from="D03" to="OP01" timestamp="2008-08-08T07:09:10"/>
</bindings>
```

Figure 4.12: A snapshot of a task run descriptor at the *Success* state.

visualization intensive tasks, or the tasks that can be executed in the host of the TYPE-A executor. A TYPE-A executor is typically deployed at a client-side machine such that a user can view and interact with the graphical user interfaces of tasks assigned to the executor. Each TYPE-A executor is required to communicate *remotely* with the Task Master and *locally* with tasks. To avoid the clutter of display, tasks are executed sequentially in an execution environment provided by a TYPE-A executor.

2) A *TYPE-B* executor provides an execution environment mostly for tasks with tasks components being Web services, whose interfaces are described by WSDL. A TYPE-B executor can be deployed either at the host of the Task Master or at any other standalone host. Each TYPE-B executor is required to communicate *remotely* with tasks, which can be executed in parallel.

3) A *TYPE-C* executor provides an execution environment for tasks that are registered and specified to execute on remote systems, including the underlying high-end computing environment, such as Grids and Clusters. Typically, those tasks require long-duration back-end computations without user interactions. This type of executors can be deployed either at the host of the Task Master or at any other standalone host. Each TYPE-C executor is required to

communicate *remotely* with tasks, which can be executed in parallel.

4) A *TYPE-D* executor provides an execution environment for built-in tasks and those that are registered and specified to execute at the host where the Task Master is deployed. Those built-in tasks can be hard-coded into the subsystem and installed with the Task Master. Each TYPE-D executor communicates *locally* with both the Task Master and tasks, and tasks can be executed in parallel.

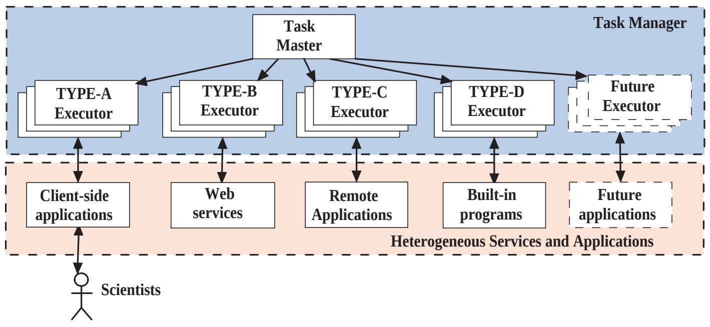


Figure 4.13: VIEW Task Manager for the execution of heterogeneous shims and functional task components.

Different types of Task Executors implement different internal functions to accommodate tasks using programming languages and invocation mechanisms, but all of them provide uniform interfaces to the Task Master on one hand and uniform interfaces to services and applications on the other hand. The architecture of Task Executors are extensible in nature: to support new types of task components in the future, it is only required for a particular Task Executor to add new functions to incorporate their invocation methods without affecting other Task Executors and the Task Master.

4.5 Case Study: a Heterogeneous Scientific Workflow for Automating Imaging Analysis

Figure 4.14.(a) shows a scientific workflow designed for automating imaging analysis of fiber tracts in human brains. The tasks in this workflow were developed by multiple research groups, so they are heterogeneous in nature and distributively deployed on various computing environments.

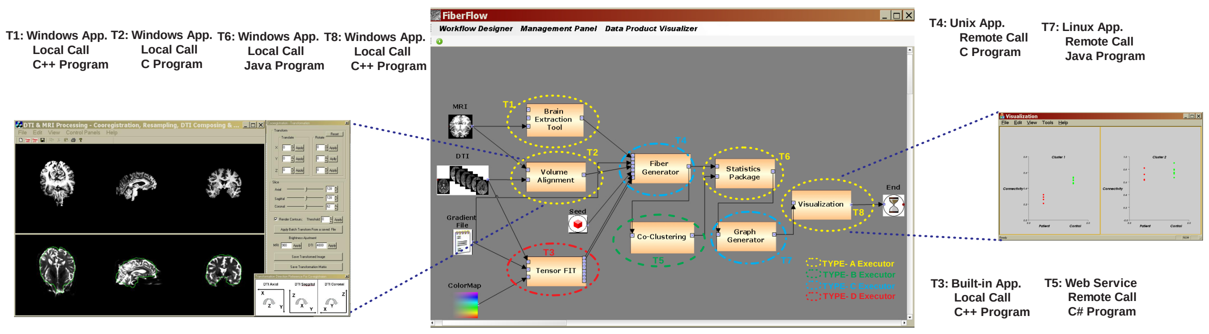

T1: Windows App. Local Call C++ Program
T2: Windows App. Local Call C Program
T6: Windows App. Local Call Java Program
T8: Windows App. Local Call C++ Program
T4: Unix App. Remote Call C Program
T7: Linux App. Remote Call Java Program
T3: Built-in App. Local Call C++ Program
T5: Web Service Remote Call C# Program
FiberFlow
Workflow Designer
Management Panel
Data Product Visualizer
MRI
DTI
Gradient File
Seed
ColorMap
Brain Extraction Tool
Volume Alignment
Tensor FIT
Fiber Generator
Co-Clustering
Statistics Package
Graph Generator
Visualization
End
TYPE-A Executor
TYPE-B Executor
TYPE-C Executor
TYPE-D Executor

Figure 4.14: A scientific workflow composed of heterogeneous applications and services.

The *Brain Extraction Tool* task (T1) strips off a subject's skull based on Magnetic Resonance Imaging (MRI) volume files; the *Volume Alignment* task (T2) generates a transformation matrix, which indicates the spatial mappings from Diffusion Tensor Imaging (DTI) volume files to MRI files; The *Statistics Package* task (T6) allows scientists to mark their interested coclustering regions and then conducts statistical analysis. The *Visualization* (T8) task visualizes statistical coclustering results. Although these four tasks are developed in various programming languages, they are local Windows applications and require intensive user-interactions; hence, they are handled by a TYPE-A executor. The *Fiber Generator* task (T4) and *Graph Generator* task (T7) are two compute-intensive tasks, so they are assigned to a TYPE-C executor and executed in the Wayne State Grid and a remote high-performance Linux server, respectively. The *Coclustering* task (T5) is exposed as a Web service, and it is developed for clustering human brain fiber tracts into different bundles. These bundles can be employed to generate statistical hypotheses and to identify particular neural disorders. T5 is assigned to a TYPE-B executor. The *Tensor Fit* task (T3) computes the tensor field using DTI, gradient File and ColorMap to generate various invariant metrics. T3 is a fundamental step needed for most preprocessing procedures, so it is built in the system and processed by a TYPE-D executor.

Supported by the Task Manager, the heterogeneity and distribution of these tasks are transparent to users when they construct this workflow. Figure 4.14 also shows T2's user interface waiting for user interaction during a task run and a final statistical result after running T8.

CHAPTER 5: WORKFLOW SCHEDULING FOR SERVICES COMPUTING ENVIRONMENTS

With the advent of services computing technologies, thousands or even millions of distributed compute resources are able to be exposed as services for other applications or services to access through the Internet. In such a services computing environment, the number of assigned resources to a workflow can be elastically scaled out or in by service requests. Even though there have been many work on workflow scheduling in the literature, most of proposed solutions address the problem of assigning a workflow to a bounded number of resources. The number of resources cannot be automatically determined on demand of the size of the workflow and these resources assigned to the workflow will not be released until the execution of the workflow completes. As a result, resources assigned to a workflow are sometimes insufficient to the execution of workflows, which leads to a long execution duration, especially for compute-intensive workflows; or many resources keep idle most of the time during the workflow execution, especially for data-intensive workflows, which leads to a waste of resources and budgets.

The ability of services computing to scale on demand as usage changes through dynamic provisioning brings a new opportunity to solve this scheduling problem; however, none of the current scheduling algorithms are applicable in such emerging services computing environments. To present our solution, we firstly introduce a services computing environment in Section 5.1 and a workflow graph representation for such an environment in Section 5.2, followed by a formalization of the workflow scheduling problem in Section 5.3. Then two workflow scheduling algorithms - the SHEFT algorithm and the SCPOR algorithm - are proposed in Section 5.4 to schedule workflows in a services computing environment, which not only optimize workflow execution time but also allow the number of requested resources to change on demand. Finally, extensive experiments and comparisons are performed to evaluate our proposed solution in Section 5.5.

5.1 Services Computing Environment

Due to the complexity of scientific processes, scientific workflows have become increasingly compute and data intensive. These scientific workflows are often required to be executed in distributed computing environments, including the recently emerged services computing environments. A services computing environment has several features that are distinct from other computing environments: (1) Compute resources in the environment are exposed as services that provide a standardized interface for other applications or services to access over the network; (2) The number and type of compute resources assigned to a workflow are determined by service requests; (3) The number of assigned resources to the workflow can be dynamically changed at runtime: if initially assigned resources are insufficient to an execution, additional resources can be assigned; if a resource keeps idle for a long time, it can be released to the environment. Therefore, workflow compute resources from such an environment can be *elastically* scaled out or in on demand; (4) Not all requested compute resources are necessary to be assigned at the beginning of the execution. Resources can be assigned only if an execution is in need.

Resources provisioned by such an environment are often heterogeneous in terms of computing capability and data communication. The computing capability of a compute resource is mainly determined by the configuration of the number of processors and capability of the processors. It means that the variation is possible among the execution times for a given task across all the machines. The resources are connected to each other by an internal network with different data transfer rates (or bandwidths). In this case, we model such an environment by partitioning all resources into a number of clusters. Resources with the same computing capability are grouped into one cluster. Resources within one cluster share the same network communication, so they have the same data transfer rate with each other within this cluster. Also, resources within one cluster share the same data transfer rate for transferring data to resources in another cluster. Therefore, a services computing environment can be defined as:

Definition 5.1.1 (Services Computing Environment $E(R_E, C_E, F_M, F_B, F_R)$)**.** A services computing environment is a 5-tuple $E(R_E, C_E, F_M, F_B, F_R)$, where

- R_E is a set of resources in the environment,
- C_E is a set of clusters that partition the resources R_E,
- $F_M : R_E \to C_E$ is the mapping function that maps a resource to its cluster number. $F_M(R_i), R_i \in R_E$ gives the cluster number $C_j, C_j \in C_E$ that R_i belongs to.
- $F_B : C_E \times C_E \to Q_0^+$ is the data communication rate function. $F_B(C_i, C_j), C_i, C_j \in C_E$ gives the data communication rate between C_i and C_i. Q_0^+ is the set of non-negative rational number.
- $F_R : R_E \to Q^+$ is the resource computing speed function. $F_R(R_i), R_i \in R_E$ gives the speed for the computing resource R_i, measured in some pre-determined unit like million instructions per machine cycles or million instructions per nanoseconds. Q^+ is the set of positive rational number.

$\square$

If the computing capability of all resources are the same, then all these resources are in the same cluster ($|C_E| = 1$). In this case, the model accommodates a homogeneous computing environment; if the computing capability of all resources are different, then each cluster only contains one resource ($|C_E| = |R_E|$). In this case, the model accommodates a completely heterogeneous computing environment.

In such a computing environment, it is assumed that each task of the workflow can be processed on any of the assigned resources. An accurate estimate of the execution time for each task on each machine is known prior to execution. The computation of tasks on each compute resource can be overlapped with data communication between resources. During workflow execution, tasks assigned on one resource can be executed in parallel with tasks on other resources; however, no task is allowed to run in parallel on two resources at the same time.

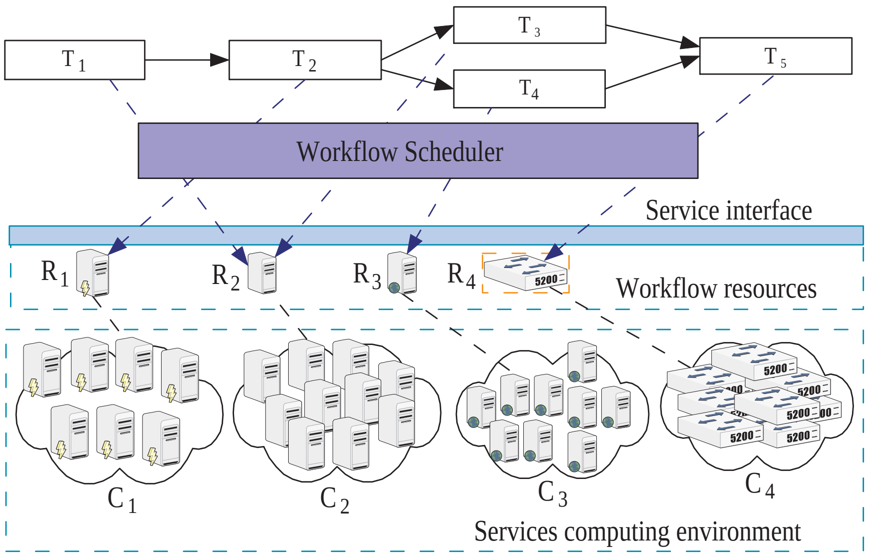


Figure 5.1: Workflow scheduling in services computing environments.

In Figure 5.1, a workflow that consists of five tasks is scheduled in a services computing environment. Inside the environment, compute resources are partitioned into four clusters. For example, high-performance computers are distributed in cluster C_1, while the computers with the lowest computing capability are connected in cluster C_2. Initially, three resources (R_1, R_2, and R_3) are assigned to the workflow, which are selected from clusters C_1, C_2 and C_3, respectively. An additional resource R_4 from cluster C_4 is assigned later by a request of the workflow scheduler. After scheduling this workflow, task T_1, as the entry task of the workflow, is firstly executed on R_2, followed by the execution of task T_2 on R_1. After that, tasks T_3 and T_4 can be executed in parallel on resources R_2 and R_3. Since both T_1 and T_3 are assigned onto R_2, T_3 is scheduled to start sometime after T_1 completes. Task T_5 can start on resource R_4 after both T_3 and T_4 complete their executions. Each task may wait for its input data transferred to the scheduled resources before the task starts to run.

5.2 Workflow Graph Representation

A scientific workflow is a computerized model of a scientific process, and it consists of a set of tasks and a set of data dependencies between these tasks. Each task in a workflow is *atomic*, so the operations of a task are not allowed to be interrupted during task execution. A task produces a dataset that can be consumed by another task of the workflow. A data dependency specifies that an amount of dataset is required to be transferred after which task completes and before which task starts. A scientific workflow can be formally defined as:

Definition 5.2.1 (Scientific Workflow $W(T, D, F_T, F_D)$)**.** A scientific workflow is a 4-tuple $W = (T, D, F_T, F_D)$, where

- T is the set of tasks in the workflow,
- $D = \{< T_i, T_j > \mid T_i, T_j \in T, i \neq j, i, j \leq |T|, T_j\, consumes\, data\, D_{i,j}\, produced\, by\, T_i\}$ is the set of data dependencies. $D_{i,j}$ denotes that an amount of data is required to be transferred after T_i completes and before T_j starts.
- $F_T : T \rightarrow Q_0^+$ is the execution cost function. $F_T(T_i), T_i \in T$ gives the execution time of a task T_i, measured in some pre-determined unit like million instructions, machine cycles or nanoseconds.
- $F_D : D \rightarrow Q_0^+$ is the data size function. $F_D(D_{i,j}), D_{i,j} \in D$ gives the size of a dataset $D_{i,j}$, measured in some pre-determined unit like bits or bytes.

$\square$

Definition 5.2.2 (Predecessors $pred(T_j)$ and Successors $succ(T_i)$)**.** Given a workflow W, if the start of a task T_j depends on the completion of a task T_i, then T_i is an immediate predecessor of T_j, and T_j is an immediate successor of T_i. The task precedence relation can be denoted as $T_i \rightarrow T_j$. The set of immediate predecessors of T_i is denoted as $pred(T_i)$, and the set of immediate successors of T_i is denoted as $succ(T_i)$. □

A task that has no any predecessors is called an *entry* task; a task that has no any successors is called an *exit* task. The entry task and exit task of the workflow are denoted as T_{entry} and T_{exit}, respectively. The execution of a workflow starts from an entry task and ends with an exit task.

Definition 5.2.3 (Scientific Workflow Graph $G(T, D, R, F_c, F_{\overline{c}}, F_p, F_{\overline{p}}, F_u, F_d, F_r)$)**.** Given a workflow $W(T, D, F_T, F_D)$ in a computing environment $E(R_E, C_E, F_M, F_B, F_R)$, a weighted directed acyclic graph that represents the workflow is a 5-tuple $G(T, D, R, F_c, F_{\overline{c}}, F_p, F_{\overline{p}}, F_u, F_d, F_r)$, where

- the vertices of the graph represent the set of tasks T,
- the edges of the graph represent the set of data dependencies D,
- R is a set of resources assigned to W, $R \in R_E$,
- $F_c : D \times R_E \times R_E \rightarrow Q_0^+$ is the data communication cost function. $F_c(i, j, m, n), D_{i,j} \in D, R_m, R_n \in R_E$ gives the data communication cost of $D_{i,j}$ from resource R_m to resource R_n.
- $F_{\overline{c}} : D \times R \rightarrow Q_0^+$ is the average data communication cost function. $F_c(i, j), D_{i,j} \in D$ gives the average data communication cost of $D_{i,j}$ in resources R, which is taken as the weight of edge in the graph G. A communication edge may have no data, if solely to enforce a precedence constraint. The data size of such an edge is assumed to be zero.
- $F_p : T \times R \rightarrow Q^+$ is the task computation cost function. $F_p(T_i, R_m), T_i \in T, R_m \in R$ gives the computation cost of T_i on resource R_m.
- $F_{\overline{p}} : T \times R \rightarrow Q^+$ is the average task computation cost function, $F_{\overline{p}}(T_i)$ gives the average computation cost of task T_i, which is taken as the weight of vertex in the graph G.

- $F_u : T \rightarrow Q^+$ is the upward rank function. $F_u(T_i)$ gives the value of task T_i's upward rank in the workflow graph.
- $F_d : T \rightarrow Q^+$ is the downward rank function. $F_d(T_i)$ gives the value of task T_i'sdownward rank in the workflow graph.
- $F_r : T \rightarrow Q^+$ is the priority rank function. $F_r(T_i)$ gives the value of task T_i's priority rank in the workflow graph.

□

If a workflow has multiple entry tasks or exit tasks, a virtual entry task or exit task will be connected to these entry tasks or exit tasks. The virtual task has zero computation cost and zero communication cost between the task and other tasks. Therefore, each workflow graph *only* has one entry task and one exit task. The in-degree of each vertex in the workflow graph G is defined as:

$$\begin{cases} d^+(T_i) = 0, if\ T_i = T_{entry}, \\ d^+(T_i) > 0, if\ T_i \neq T_{entry}, T_i \in T. \end{cases} \tag{5.1}$$

The out-degree of each vertex in the workflow graph is defined as:

$$\begin{cases} d^-(T_i) = 0, if\ T_i = T_{exit} \\ d^-(T_i) > 0, if\ T_i \neq T_{exit}, T_i \in T. \end{cases} \tag{5.2}$$

Example 5.2.4. Figure 5.2 illustrates a workflow graph G that consists of 14 tasks, in which T_1 is the entry task and T_{14} is the exit task of the workflow. $T = \{T_1, T_2, \cdots, T_{14}\}$, $|T| = 14$, $D = \{D_{1,2}, D_{1,3}, D_{1,4}, D_{1,5}, D_{1,6}, D_{1,7}, D_{2,8}, D_{3,9}, D_{4,8}, D_{4,10}, D_{5,9}, D_{5,11}, D_{6,10}, D_{7,11}, D_{8,12}, D_{9,13}, D_{10,12}, D_{11,13}, D_{12,14}, D_{13,14}\}$. $pred(T_1) = \emptyset$, $succ(T_1) = \{T_2, T_3, T_4, T_5, T_6, T_7\}, pred(T_9) = \{T_3, T_5\}, succ(T_9) = \{T_{13}\}, pred(T_{14}) = \{T_{12}, T_{13}\}$, and $succ(T_{14}) = \emptyset$. □

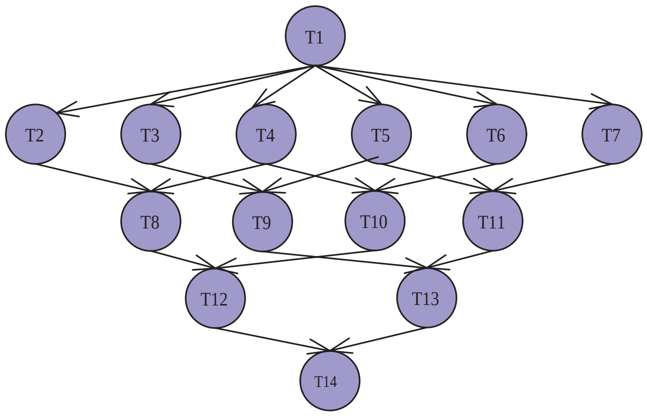


Figure 5.2: An example of a workflow graph.

Definition 5.2.5 (Communication Cost F_c and Average Communication Cost $F_{\overline{c}}$)**.** Given a workflow $W(T, D, F_T, F_D)$ in a computing environment $E(R_E, C_E, F_M, F_B, F_R)$, data $D_{i,j} \in D$ is transferred from task $T_i \in T$ on resource $R_m \in R_E$ to task $T_j \in T$ on resource $R_n \in R_E$. The communication cost of $D_{i,j}$ in a workflow graph G is defined as:

$$F_c(i,j,m,n) = \begin{cases} 0, \ if \ m = n, \\ \frac{F_D(D_{i,j})}{F_B(F_M(R_m),F_M(R_n))}, \ if \ F_M(R_m) \neq F_M(R_n). \end{cases} \tag{5.3}$$

If T_i and T_j are mapped onto the same resource $(m = n)$, then $F_c(i, j, m, n) = 0$. If T_i and T_j are mapped onto different resources $(F_M(R_m) \neq F_M(R_n))$, the data communication cost are calculated using data transfer rate between cluster $F_M(R_m)$ and $F_M(R_n)$.

Given a set of clusters C that are assigned to the workflow, $C = \{c | \forall R_i \in R, \exists c = F_M(R_i), c \in C_E\}$. The average data transfer rate among all resources R can be defined as:

$$\overline{B} = \frac{|C|}{\sum_{C_i,C_j \in C} F_B(C_i, C_j)}. \tag{5.4}$$

The average communication cost of $D_{i,j}$ is defined as:

$$F_{\overline{c}}(i,j) = \frac{F_D(D_{i,j})}{\overline{B}}. \tag{5.5}$$

□

Example 5.2.6. According to Table 5.1.(b), the data transfer rate between clusters C_1 and C_2 is 11, between C_1 and C_3 is 28, and between C_2 and C_3 is 26. From Table 5.1.(c), it is known that R_1, R_2 and R_3 are selected from clusters C_1, C_2 and C_3. The total number of clusters assigned to this workflow is equal to 3. In this case, the average data transfer rate between R_1, R_2 and R_3 is $\overline{B} = 3/(1/11 + 1/28 + 1/26) \approx 18.1725$. From Table 5.1.(a), the data size of $D_{1,2}$ is equal to 229, so the average communication cost of $D_{1,2}$ between R_1, R_2 and R_3 is $F_{\overline{c}}(1,2) = 229/18.1725 \approx 12.601$. The rest of the average data communication costs are calculated in a similar way. □

Definition 5.2.7 (Computation Cost F_p and Average Computation Cost $F_{\overline{p}}$)**.** Given a workflow $W(T, D, F_T, F_D)$ in a computing environment $E(R_E, C_E, F_M, F_B, F_R)$, the computation cost of a task $T_i \in T$ on a resource $R_m \in R_E$ in a workflow graph G is defined as:

$$F_p(T_i, R_m) = \frac{F_T(T_i)}{F_R(R_m)}. \tag{5.6}$$

The average computation cost of a task T_i in resources R is defined as:

$$F_{\overline{p}}(T_i) = \frac{\sum_{m=1}^{|R|} F_p(T_i, R_m)}{|R|}, \tag{5.7}$$

□

Example 5.2.8. Table 5.2 presents the computation costs for the workflow in Figure 5.2. Resource R_1, R_2, and R_3 are assigned to the workflow and the computation costs of each task

$<T_i,T_j>$	$F_D(D_{i,j})$	$F_{\overline{p}}(T_i)$
$<T_1,T_2>$	229	12.601
$<T_1,T_3>$	103	5.668
$<T_1,T_4>$	97	5.338
$<T_1,T_5>$	66	3.632
$<T_1,T_6>$	19	1.046
$<T_1,T_7>$	95	5.228
$<T_2,T_8>$	26	1.431
$<T_3,T_9>$	48	2.641
$<T_4,T_8>$	245	13.482
$<T_4,T_{10}>$	188	10.345
$<T_5,T_{10}>$	215	11.831
$<T_5,T_{11}>$	225	12.381
$<T_6,T_{10}>$	246	13.537
$<T_7,T_{11}>$	10	0.550
$<T_8,T_{12}>$	211	11.611
$<T_9,T_{13}>$	71	3.907
$<T_{10},T_{12}>$	26	1.431
$<T_{11},T_{13}>$	120	6.603
$<T_{12},T_{14}>$	125	6.879
$<T_{13},T_{14}>$	215	11.831

(a)

C_m	C_n	$F_B(C_m, C_n)$
C_1	C_2	11
C_2	C_1	11
C_1	C_3	28
C_3	C_1	28
C_2	C_3	26
C_3	C_2	26
...	...	...

(b)

R_m	$F_M(R_m)$
R_1	C_1
R_2	C_2
R_3	C_3
R_4	C_1
R_5	C_3
R_6	C_1
...	...

(c)

Table 5.1: (a) The average communication costs of the workflow in Figure 5.2; (b) the data transfer rates between clusters; (c) The list of mappings between resources and clusters.

on these three resources are different. For example, the computation costs of T_1 on R_1, R_2, and R_3 are $6, 18$ and 9, respectively. The average computation cost $F_{\overline{p}}(T_1) = (6 + 18 + 9)/3 = 11.000$. The average computation costs of other tasks are calculated in a similar way. □

The upward rank $F_u(T_i)$ is computed recursively by traversing the workflow graph upward, starting from T_{exit}. It measures the longest path of the workflow graph from T_{exit} to T_i, which is the sum of the average computation cost of T_i, and the longest path from T_{exit} to the successors of T_i, determined by the sum of $F_u(T_j), T_j \in succ(T_i)$ and the data communication cost between task T_i and T_j.

Definition 5.2.9 (Upward Rank $F_u(T_i)$)**.** Given a workflow $W(T, D, F_T, F_D)$ in a computing environment $E(R_E, C_E, F_M, F_B, F_R)$, a task T_i's upward rank in a workflow graph G is defined

T_i	R_1	R_2	R_3	$F_{\overline{p}}(T_i)$
T1	6	18	9	11.000
T2	13	24	17	18.000
T3	16	20	23	19.667
T4	11	13	5	9.667
T5	13	23	15	17.000
T6	29	10	29	22.667
T7	12	26	25	21.000
T8	24	13	12	16.333
T9	18	10	9	12.333
T10	26	22	16	21.333
T11	23	26	22	23.667
T12	11	24	10	15.000
T13	12	22	10	14.667
T14	15	23	12	16.667

Table 5.2: Computation costs of the workflow in Figure 5.2.

as:

$$F_u(T_i) = \begin{cases} F_{\overline{p}}(T_i),\ if\ T_i = T_{exit}, \\ F_{\overline{p}}(T_i) + \max_{T_j \in succ(T_i)}(F_{\overline{c}}(i,j) + F_u(T_j)),\ otherwise. \end{cases} \tag{5.8}$$

□

Example 5.2.10. Below, we demonstrate the upward rank computation procedure for each task of the workflow in Figure 5.2, given the computation costs and communication costs in Table 5.2 and Table 5.1.

$F_u(T_{14}) = F_{\overline{p}}(T_{exit}) \approx 14.667,$

$F_u(T_{13}) = F_{\overline{p}}(T_{13}) + \max\{F_{\overline{c}}(13,14) + F_u(T_{14}) = 14.6667 + +\max\{11.831 + 16.6667\} = 43.164,$

$F_u(T_{12}) = F_{\overline{p}}(T_{12}) + \max\{F_{\overline{c}}(12,14) + F_u(T_{14})\} = 15.0000 + \max\{6.879 + 16.6667\} = 15.0000 + 23.5454 \approx 38.545,$

$F_u(T_{11}) = F_{\overline{p}}(T_{11}) + \max\{F_{\overline{c}}(11,13) + F_u(T_{13})\} = 23.6667 + \max\{6.603 + 38.545\} = 23.6667 + 45.148 \approx 73.434,$

$F_u(T_{10}) = F_{\overline{p}}(T_{10}) + \max\{F_{\overline{c}}(10,12) + F_u(T_{12})\} = 21.3333 + \max\{1.431 + 38.545\} = 21.3333 + 39.976 \approx 61.309$,

$F_u(T_9) = F_{\overline{p}}(T_9) + \max\{F_{\overline{c}}(9,13) + F_u(T_{13})\} = 12.3333 + \max\{3.907 + 43.164\} = 12.3333 + 47.071 \approx 59.405$,

$F_u(T_8) = F_{\overline{p}}(T_8) + \max\{F_{\overline{c}}(8,12) + F_u(T_{12})\} = 16.3333 + \max\{11.611 + 38.545\} = 16.3333 + 50.156 \approx 66.490$,

$F_u(T_7) = F_{\overline{p}}(T_7) + \max\{F_{\overline{c}}(7,11) + F_u(T_{11})\} = 21.0000 + \max\{0.550 + 61.309\} = 21.0000 + 61.859 \approx 94.985$,

$F_u(T_6) = F_{\overline{p}}(T_6) + \max\{F_{\overline{c}}(6,10) + F_u(T_{10})\} = 22.6667 + \max\{13.537 + 61.309\} = 22.6667 + 74.846 \approx 97.513$,

$F_u(T_5) = F_{\overline{p}}(T_5) + \max\{F_{\overline{c}}(5,9) + F_u(T_9)), (F_{\overline{c}}(5,11) + F_u(T_{11}))\} = 17.0000 + \max\{(11.831 + 59.405), (12.381 + 73.434)\} = 17.0000 + 85.815 \approx 102.816$,

$F_u(T_4) = F_{\overline{p}}(T_4) + \max\{(F_{\overline{c}}(4,8) + F_u(T_8)), (F_{\overline{c}}(4,10) + F_u(T_{10}))\} = 9.6667 + \max\{(13.482 + 66.490), (10.345 + 61.309)\} = 9.6667 + 79.972 \approx 89.638$,

$F_u(T_3) = F_{\overline{p}}(T_3) + \max\{F_{\overline{c}}(3,9) + F_u(T_9)\} = 19.6667 + \max\{2.641 + 59.405\} = 19.6667 + 62.046 \approx 81.713$,

$F_u(T_2) = F_{\overline{p}}(T_2) + \max\{F_{\overline{c}}(2,8) + F_u(T_8)\} = 18.0000 + \max\{1.431 + 66.490\} = 18.0000 + 67.921 \approx 85.920$,

$F_u(T_1) = F_{\overline{p}}(T_1) + \max\{(F_{\overline{c}}(1,2) + F_u(T_2)), (F_{\overline{c}}(1,2) + F_u(T_3)), (F_{\overline{c}}(1,3) + F_u(T_3)), (F_{\overline{c}}(1,4) + F_u(T_4)), (F_{\overline{c}}(1,5) + F_u(T_5)), (F_{\overline{c}}(1,5) + F_u(T_6)), (F_{\overline{c}}(1,6) + F_u(T_7)), (F_{\overline{c}}(1,7) + F_u(T_7))\} = 11.0000 + \max\{(12.601 + 85.920), (5.668 + 81.713), (5.338 + 89.638), (3.632 + 102.816), (1.046 + 97.513), (5.228 + 94.985)\} = 11.0000 + 106.448 \approx 117.448$. $\square$

The downward rank $F_d(T_i)$ measures the longest path from T_{entry} to T_i, determined by the computation cost of T_i, the data communication costs between T_i and its predecessors, and the F_d value of T_i's predecessors.

Definition 5.2.11 (Downward Rank $F_d(T_i)$)**.** Given a workflow $W(T, D, F_T, F_D)$ in a computing environment $E(R_E, C_E, F_M, F_B, F_R)$, a task T_i's downward rank in a workflow graph G is defined as:

$$F_d(T_i) = \begin{cases} 0, if\ T_i = T_{entry}, \\ \max_{T_j \in pred(T_i)}(F_{\overline{p}}(T_j) + F_{\overline{c}}(j,i) + F_d(T_j)),\ otherwise. \end{cases} \tag{5.9}$$

□

Example 5.2.12. Below, we demonstrate the downward rank computation procedure for each task of the workflow in Figure 5.2, given the computation costs and communication costs in Table 5.2 and Table 5.1.

$F_d(T_1) = 0,$

$F_d(T_2) = \max(F_{\overline{p}}(T_1) + F_{\overline{c}}(1,2) + F_d(T_1)) = \max(11.000 + 12.601 + 0) \approx 23.601,$

$F_d(T_3) = \max(F_{\overline{p}}(T_1) + F_{\overline{c}}(1,3) + F_d(T_1)) = \max(11.000 + 5.668 + 0) \approx 16.668,$

$F_d(T_4) = \max(F_{\overline{p}}(T_1) + F_{\overline{c}}(1,4) + F_d(T_1)) = \max(11.000 + 5.338 + 0) \approx 16.338,$

$F_d(T_5) = \max(F_{\overline{p}}(T_1) + F_{\overline{c}}(1,5) + F_d(T_1)) = \max(11.000 + 3.632 + 0) \approx 14.632,$

$F_d(T_6) = \max(F_{\overline{p}}(T_1) + F_{\overline{c}}(1,6) + F_d(T_1)) = \max(11.000 + 1.046 + 0) \approx 12.046,$

$F_d(T_7) = \max(F_{\overline{p}}(T_1) + F_{\overline{c}}(1,7) + F_d(T_1)) = \max(11.000 + 5.228 + 0) \approx 16.228,$

$F_d(T_8) = \max\{(F_{\overline{p}}(T_2)+F_{\overline{c}}(2,8)+F_d(T_2)), (F_{\overline{p}}(T_4)+F_{\overline{c}}(4,8)+F_d(T_4))\} = \max\{(18.000+ 1.431 + 23.601), (9.667 + 13.482 + 16.338)\} \approx 43.032,$

$F_d(T_9) = \max\{(F_{\overline{p}}(T_3)+F_{\overline{c}}(3,9)+F_d(T_3)), (F_{\overline{p}}(T_5)+F_{\overline{c}}(5,9)+F_d(T_5))\} = \max\{(19.6667+ 2.641 + 16.668), (17.0000 + 11.831 + 14.632)\} \approx 43.463,$

$F_d(T_{10}) = \max\{(F_{\overline{p}}(T_4)+F_{\overline{c}}(4,10)+F_d(T_4)), (F_{\overline{p}}(T_6)+F_{\overline{c}}(6,10)+F_d(T_6))\} = \max\{(9.6667+ 10.345 + 16.338), (22.6667 + 13.537 + 12.046)\} \approx 48.249,$

$F_d(T_{11}) = \max\{(F_{\overline{p}}(T_5)+F_{\overline{c}}(5,11)+F_d(T_5)), (F_{\overline{p}}(T_7)+F_{\overline{c}}(7,11)+F_d(T_7))\} = \max\{(17.0000+ 12.381 + 14.632), (21.0000 + 0.550 + 16.228)\} \approx 44.013,$

$F_d(T_{12}) = \max\{(F_{\overline{p}}(T_8)+F_{\overline{c}}(8,12)+F_d(T_8)), (F_{\overline{p}}(T_{10})+F_{\overline{c}}(10,12)+F_d(T_{10}))\} = \max\{(16.3333+ 11.611 + 43.032), (21.3333 + 1.431 + 48.249)\} \approx 71.013,$

T_i	$F_u(T_i)$	$F_d(T_i)$	$F_r(T_i)$
T_1	117.448	0.000	**117.448**
T_2	85.920	23.601	109.522
T_3	81.713	16.668	98.381
T_4	89.638	16.338	105.976
T_5	102.816	14.632	**117.448**
T_6	97.513	12.046	109.558
T_7	94.985	16.228	111.212
T_8	66.490	43.032	109.522
T_9	59.405	43.463	102.868
T_{10}	61.309	48.249	109.558
T_{11}	73.434	44.013	**117.448**
T_{12}	38.545	71.013	109.558
T_{13}	43.164	74.283	**117.448**
T_{14}	16.667	100.781	**117.448**

Table 5.3: Task priority ranks for the workflow in Figure 5.2

$F_d(T_{13}) = \max\{(F_{\overline{p}}(T_9) + F_{\overline{c}}(9,13) + F_d(T_9)), (F_{\overline{p}}(T_{11}) + F_{\overline{c}}(11,13) + F_d(T_{11}))\}$

$= \max\{(12.3333 + 3.907 + 43.463), (23.6667 + 6.603 + 44.013)\} \approx 74.283,$

$F_d(T_{14}) = \max\{(F_{\overline{p}}(T_{12}) + F_{\overline{c}}(12,14) + F_d(T_{12})), (F_{\overline{p}}(T_{13}) + F_{\overline{c}}(14,13) + F_d(T_{13}))\} = \max\{(15.0000 + 6.879 + 71.013), (14.6667 + 11.831 + 74.283)\} \approx 100.781.$

□

Definition 5.2.13 (Priority Rank $F_r(T_i)$)**.** Given a workflow $W(T, D, F_T, F_D)$ in a computing environment $E(R_E, C_E, F_M, F_B, F_R)$, a task T_i's priority rank in a workflow graph G is defined as:

$$F_r(T_i) = F_u(T_i) + F_d(T_i). \tag{5.10}$$

□

5.3 Workflow Scheduling Problem Description

In order to present our solution, we firstly introduce several notions as follows.

Definition 5.3.1 (Earliest Ready Time $ERT(T_i, R_m)$)**.** Given a workflow graph $G(T, D, R, F_c, F_{\overline{c}}, F_p, F_{\overline{p}}, F_u, F_d, F_r)$, the earliest start time of T_i is the earliest time when all predecessors have completed their executions and all input data have arrived at a resource R_m, which can be defined as:

$$ERT(T_i, R_m) = \begin{cases} 0,\ if\ T_i = T_{entry}, \\ \max_{T_k \in pred(T_i)}\{EFT(T_k, R_n) + F_c(k, i, n, m)\}, if\ R_m, R_n \in R, \\ T_i \neq T_{entry}, \\ \max_{T_k \in pred(T_i)}\{EFT(T_k, R_n) + F_c(k, i, n, m\prime)\},\ if\ R_m, R_n \in R_E \setminus R, \\ \exists R_{m\prime} \in R, F_M(R_{m\prime}) = F_M(R_m), \\ \max_{T_k \in pred(T_i)}\{EFT(T_k, R_n) + F_{\overline{c}}(k, i)\},\ if\ R_m, R_n \in R_E \setminus R, \\ \forall R_{m\prime} \in R, F_M(R_{m\prime}) \neq F_M(R_m). \end{cases} \tag{5.11}$$

□

If an entry task T_i is scheduled onto a resource $R_m \in R$, then $ERT(T_i, R_m) = 0$; Otherwise, three cases are considered according to the resource R_m that T_i is assigned to: (1) if R_m is a resource that has already been assigned to the workflow ($R_m \in R$), then $ERT(T_i, R_m)$ is determined by the earliest time when all predecessors of T_i have completed executions and all the input data of T_i have transferred from R_n to R_m; (2) if R_m is a resource from the environment that has not been assigned to the workflow ($R_E \backslash R$) and its computing capability matches at least one of the resources ($\exists R_{m\prime} \in R, F_M(R_{m\prime}) = F_M(R_m)$), it means that the execution time of T_i on R_m and $R_{m\prime}$ are the same. The communication cost from T_k to T_i on R_m is assumed to be equal to the communication cost from T_k to T_i on $R_{m\prime}$. The new resource R_m is available once it is assigned to T_i, and after T_i completes execution, R_m is available to other tasks of the workflow ($R = R \cup R_m$); (3) if R_m is a new resource that could be assigned to the workflow ($R_m \notin R$) but its computing capability does not match any of the currently assigned resources $(\forall R_{m'} \in R, F_M(R_{m'}) \neq F_M(R_m)$, then the communication cost from T_k to T_i is estimated by the average communication cost $F_{\overline{c}}(k,i)$. $EFT(T_k, R_n)$ is the earliest finish time of task T_i's predecessor T_k on its assigned resource R_n. A task's earliest finish time is defined as follows.

Definition 5.3.2 (Earliest Finish Time $EFT(T_i, R_m)$)**.** Given a a workflow graph $G(T, D, R, F_c, F_{\overline{c}}, F_p, F_{\overline{p}}, F_u, F_d, F_r)$, if task T_i is scheduled onto a resource R_m, the earliest finish time of T_i is defined as:

$$EFT(T_i, R_m) = \begin{cases} F_{\overline{p}}(T_i) + EST(T_i, R_m),\ if\ R_m, R_n \in R_E \setminus R, \forall R_{m'} \in R, \\ F_M(R_{m'}) \neq F_M(R_m), \\ F_p(T_i, R_m) + EST(T_i, R_m),\ otherwise. \end{cases} \tag{5.12}$$

□

Once a task T_i is scheduled on a resource R_m, the earliest finish time of T_i on R_m is assigned to the *task finish time* of task T_i, denoted as $TFT(T_i)$. $EST(T_i, R_m)$ is the earliest time of task T_i on a resource R_m, which is defined as follows.

Definition 5.3.3 (Earliest Start Time $EST(T_i, R_m)$)**.** Given a workflow graph $G(T, D, R, F_c, F_{\overline{c}}, F_p, F_{\overline{p}}, F_u, F_d, F_r)$, the earliest start time of T_i on a resource R_m can be defined as:

$$EST(T_i, R_m) = \begin{cases} ERT(T_i, R_m),\ if\ T_i = T_{entry}, \\ \max\{getAvailTime(R_m), ERT(T_i, R_m)\}, otherwise. \end{cases} \tag{5.13}$$

where $getAvailTime(R_m)$ returns the earliest time that R_m is ready for a task execution. □

Example 5.3.4. Given the workflow in Figure 5.2, computation costs in Table 5.2, and communication costs in Table 5.1, (1) suppose T_1 (the entry task) is scheduled on R_1, $EST(T_1, R_1) = 0$, $EFT(T_1, R_1) = F_p(T_1, R_1) + EST(T_1, R_1) = 6 + 0 = 6$; (2) Suppose T_5 is the next task scheduled on resource $R_1 \in R$, $EST(T_5, R_1) = \max\{getAvailTime(R_1), ERT(T_1, R_1) + F_c(1,5,1,1)\}$. As T_1 has already scheduled on R_1, and $EFT(T_1, R_1) = 6$, the earliest available time for R_1 is equal to 6. The data communication cost between T_1 and T_5 is zero since they are assigned to the same resource. In this case, $EST(T_5, R_1) = \max\{6, (6+0)\} = 6$, $EFT(T_5, R_1) = F_p(T_5, R_1) + EST(T_5, R_1) = 13 + 6 = 19$; (3) Suppose T_5 is scheduled on resource $R_4 \notin R$, $F_M(R_4) = F_M(R_3), R_4, R_3 \in C_3$, the data transfer rate between C_1 and C_3 is equal to 28, so the data communication cost between between T_1 on R_1 and T_5 on R_4 is $F_c(1,5,1,4) = F_D(D_{1,5})/B_{1,3} = 66/28 \approx 2.357$. $EST(T_5, R_4) = ERT(T_1, R_1) + F_c(1,5,1,4) = 6 + 2.357 = 8.357$, and $EFT(T_5, R_4) = F_p(T_5, R_4) + EST(T_5, R_4) = F_p(T_5, R_3) + EST(T_5, R_4) = 15 + 8.357 = 23.357$; (4) Suppose T_5 is scheduled on resource $R_7 \notin R$,$\forall R_{m\prime} \in R, F_M(R_7) \neq F_M(R_{m\prime}), R_7 \in C_4$, the data communication cost between T_1 on R_1 and T_5 on R_7 is $F_c(1,5,1,7) \approx F_{\overline{c}}(1,5) = 66/18.1725 \approx 3.632$. In this case, $EST(T_5, R_7) = ERT(T_5, R_7) + F_{\overline{c}}(1,5) = 9.632$, and $EFT(T_5, R_7) \approx 26.632$. □

Definition 5.3.5 (Workflow makespan WMS)**.** Given a workflow graph G in a computing environment $E(R_E, C_E, F_M, F_B, F_R)$, the total completion time of the workflow, denoted as WMS, is defined as:

$$WMS = TFT(T_{exit}). \tag{5.14}$$

□

Finally, the scheduling problem in a services computing environment can be formally stated as follows. Given a workflow $W(T, D, F_T, F_D)$ in a computing environment $E(R_E, C, F_M, F_B, F_R)$, the workflow can be represented by a weighted directed acyclic graph $G(T, D, R, F_c, F_{\overline{c}}, F_p, F_{\overline{p}}, F_u, F_d, F_r)$. A workflow schedule is required to map all tasks of this workflow to the assigned resources and order the execution of these tasks on the resources, such that WMS can be minimized.

5.4 Proposed Scheduling Algorithms

Our solution to the scheduling problem consists of two phases: a *task prioritizing* phase and a *resource selection* phase. In the task prioritizing phase, we propose a task prioritizing algorithm to rank the order of tasks for a workflow execution. In the resource selection phase, we propose a SHEFT algorithm (Scalable-Heterogeneous-Earliest-Finish-Time algorithm) and a SCPOR algorithm (Scalable-Critical-Path-On-a-Resource algorithm) to schedule large-scale workflows in a services computing environment. The SHEFT algorithm and the SCPOR algorithm are extensions of the HEFT algorithm (Heterogeneous-Earliest-Finish-Time algorithm) and the CPOP algorithm (Critical-Path-On-a-Processor) [94], which were applied for mapping a workflow application to a bounded number of processors. The detailed strategy and procedure are described as follows.

5.4.1 The Task Prioritizing Algorithm

In the task prioritizing phase, each task of a workflow is ordered by its priority rank $F_r(T_i)$. The algorithm to form a list of prioritized tasks, denoted as $List_{Priority}$, is shown in Figure 5.3. Firstly, we calculate the average communication cost $F_{\overline{c}}(i,j)(T_j \in succ(T_i))$ and computation cost $F_{\overline{p}}(T_i)$ for each task in the given workflow graph. After that, the upward rank, downward rank and priority rank for each task are calculated. Then, we create a temporary list called $List_{Candidates}$ to save tasks that are ready to be prioritized. The entry task is initially contained in the list (line 2), which means that the task is the first one to be processed. Since the entry task has no predecessors and its priority value is so far the highest in $List_{Candidates}$, the entry task is removed from $List_{Candidates}$ and added into $List_{Priority}$. Then the successors of the entry task $succ(T_{entry})$ are put into $List_{Candidates}$. From tasks in $List_{Candidates}$, we select a task T_C with the highest priority rank, remove it from $List_{Candidates}$, and then add it into $List_{Priority}$ (line 3-8). Next, for each successor of the newly removed task T_C, if all its predecessors are in $List_{Priority}$, which means that all the predecessors have already been processed, then the task is eligible to be selected into $List_{Candidates}$ (line 11-13). Such a procedure is repeated until $List_{Candidates}$ is empty (line 3 - 15). In this case, all tasks in this workflow are ranked in $List_{Priority}$.

```
    The Task Prioritizing Algorithm
    Input: $G(T, D, R, F_c, F_{\overline{c}}, F_p, F_{\overline{p}}, F_u, F_d, F_r)$
    Output: $List_{Priority}$
(1)  Begin
(2)  $List_{Priority} = \emptyset$, $List_{Candidates} = \{T_{entry}\}$
(3)  while $(List_{Candidates} \neq \emptyset)$ do
(4)      int $maxPriority = 0, T_C = NULL$
(5)      for each $(T_j \in List_{Candidates})$
(6)          if $(F_r(T_j) > maxPriority)$
(7)              then $maxPriority = F_r(T_j), T_C = T_j$
(8)      end for
(9)      $List_{Priority} = List_{Priority} \cup T_C$
(10)     $List_{Candidates} = List_{Candidates} \setminus T_C$
(11)     for each $(T_j \in succ(T_C))$
(12)         if $(pre(T_j) \subset List_{Priority})$
(13)             then $List_{Candidates} = List_{Candidates} \cup T_j$
(14)     end for
(15) end while
(16) End Function
```

Figure 5.3: The pseudo-code of the task prioritizing algorithm

Example 5.4.1. For each task of the workflow in Figure 5.2, its upward rank, downward rank and priority rank are listed in Table 5.3. To form a $List_{Priority}$ for this workflow, task T_1, as the entry task, is firstly added into $List_{Priority}$, $List_{Priority} = \{T_1\}$. Then $List_{Candidates} = succ(T_1) = \{T_2, T_3, T_4, T_5, T_6, T_7\}$. The next task will be selected from $List_{Candidates}$. As $F_r(T_5) = 117.448$ is the largest priority rank, T_5 is selected right after T_1. Now, $List_{Priority} = \{T_1, T_5\}$ and $List_{Candidates} = \{T_2, T_3, T_4, T_6, T_7\}$. According to their priority ranks, T_7 wins the third place ($List_{Priority} = \{T_1, T_5, T_7\}$). At this time, T_{11}, as an immediate successor of T_5 and T_7, is added into $List_{Candidates}$. The selection is therefore taken from $\{T_2, T_3, T_4, T_6, T_{11}\}$, their priority values are $109.522, 98.381, 105.976, 109.558$ and 117.448, respectively. In this case, T_{11} is selected into $List_{Priority}$. The rest of tasks are prioritized in a similar way. At last, the priority list of the workflow is ranked as:

$\{T_1, T_5, T_7, T_{11}, T_6, T_2, T_4, T_{10}, T_8, T_{12}, T_3, T_9, T_{13}, T_{14}\}$. $\square$

5.4.2 The SHEFT Algorithm

In the resource selection phase, tasks in a workflow are scheduled one by one according to their orders in $List_{Priority}$. Each task is mapped to a suitable resource that may minimize the earliest finish time of the task. Once a resource R_k is assigned to a task T_i, $< T_i, R_k, EST(T_i, R_k), EFT(T_i, R_k) >$ is recorded into a result set $List_{Schedule}$, which is used to instruct the workflow to be executed during runtime.

In the SHEFT algorithm (Figure 5.4), the available time of each resource $R_k \in R$ is firstly initialized to zero (line 2-4). It means that any of resources is available to be mapped to a task at the beginning of the scheduling procedure. Then a task T_i with the highest priority rank from $List_{Priority}$ is selected to be scheduled (line 7). For each resource $R_k \in R$, the earliest start time $EST(T_i, R_k)$ and earliest finish time $EFT(T_i, R_k)$ of T_i on a resource R_k is calculated. A resource that produces the smallest earliest finish time ($minEFT$) is assigned to a temporary variable R_S (line 8-12).

Next, the scheduling decision is based on the following three cases: (1) If at least one resource's available time is earlier than the minimum of task T_i's earliest ready time on a resource R_n, then T_i is mapped to R_S, $< T_i, R_S, EST(T_i, R_S), EFT(T_i, R_S) >$ is added into $List_{Schedule}$, and the available time of R_S is reset to $minEFT$ (line 13-14); (2) If none of the assignments is available at the earliest ready time of T_i, it means that all assigned resources are still in process for other tasks. In this case, a new resource R_n from the computing environment will be considered to be assigned to the workflow. If $EFT(T_i, R_n)$ is earlier than $minEFT$, then T_i is mapped to R_n, $< T_i, R_n, EST(T_i, R_n), EFT(T_i, R_n) >$ is added into $List_{Schedule}$. The resource R_n is assigned to R_S and added into R, so R_n can be reused by any other resources after $EFT(T_i, R_n)$ (line 16-19); (3) If the earliest finish time on any of new resources is later than $minEFT$, then T_i is mapped to R_S, $< T_i, R_S, EST(T_i, R_S), EFT(T_i, R_S) >$ is added

```
    The SHEFT Algorithm
    Input: $G(T, D, R, F_c, F_{\bar{c}}, F_p, F_{\bar{p}}, F_u, F_d, F_r)$,$E(R_E, C_E, F_M, F_B, F_R)$, $List_{Priority}, t_{idle}$
    Output: $List_{Schedule}$
(1)Begin
(2)   for each $R_k \in R$
(3)       $setAvailTime(R_k) = 0$
(4)   end for
(5)   while $(List_{Priority} \neq \emptyset)$
(6)       int $minEFT = MAX\_NUMBER, R_S = NULL,$
(7)       select $T_i \in List_{Priority}$ with the highest priority rank
(8)       for each $R_k \in R$
(9)           calculate $EST(T_i, R_k)$ and $EFT(T_i, R_k)$
(10)          if $(EFT(T_i, R_k) < minEFT)$
(11)              then $minEFT = EFT(T_i, R_k), R_S = R_k$
(12)      end for
(13)      if $(\min_{R_k \in R}\{getAvailTime(R_k)\} \leq \min_{R_n \in R}\{ERT(T_i, R_n)\})$
(14)          then map $T_i$ to $R_S$, update $List_{Schedule}$, $setAvailTime(R_S) = minEFT$
(15)      else
(16)          request a new resource $R_n \notin R, \exists R_{n'} \in R$, $F_M(R_n) == F_M(R_{n'})$
(17)          calculate $EST(T_i, R_n)$ and $EFT(T_i, R_n)$
(18)          if $(EFT(T_i, R_n) < minEFT)$
(19)              then map $T_i$ to $R_n$ ,$R_S = R_n, R = R_n \cup R$, update $List_{Schedule}$,
                    $minEFT = EFT(T_i, R_n), setAvailTime(R_n) = minEFT$
(20)          else
(21)              map $T_i$ to $R_S$, update $List_{Schedule}$, $setAvailTime(R_S) = minEFT$
(22)      for each $R_k \in R$
(23)          if $((minEFT - getAvailTime(R_k)) > t_{idle})$
(24)              then $R = R \setminus R_k$
(25)      end for
(26)      $List_{Priority} = List_{Priority} \setminus T_i$
(27)  end while
(28)  End Algorithm
```

Figure 5.4: The pseudo-code of the SHEFT algorithm

into $List_{Schedule}$, and the available time of R_S is reset to $minEFT$ (line 21).

For each resource R_k, if the resource R_k has been kept idle longer than a given threshold t_{idle} by the earliest finish time of task T_i ($(minEFT - getAvailTime(R_k)) > t_{idle}$), then R_k will be removed from the assigned resources R (line 22-25). There are mainly three reasons that might cause the workflow resources to scale in: (1) The initially assigned resources are more than sufficient for the workflow execution; (2) The computing capability of a resource is far less than any other assigned resources, so it has no chance for being selected; (3) An unbalanced workflow graph leads to uneven resource usage. For instance, a large number of resources are only required at the beginning of the execution. The resource availability times are updated after each task is mapped.

Example 5.4.2. According to the order in $List_{Priority}$ in Example 5.4.1, T_1 is the first task to be scheduled. At the beginning, $getAvailTime(R_1) = 0, getAvailTime(R_2) = 0, getAvailTime(R_3) = 0, R = \{R_1, R_2, R_3\}$. As an entry task, the earliest start times of T_1 on resource R_1, R_2 and R_3 are as follows: $EST(T_1, R_1) = 0, EST(T_1, R_2) = 0, EST(T_1, R_3) = 0$, the earliest finish times of T_1 on these three resources are as follows: $EFT(T_1, R_1) = F_p(T_1, R_1) + EST(T_1, R_1) = 6 + 0 = 6$, $EFT(T_1, R_2) = F_p(T_1, R_2) + EST(T_1, R_2) = 18 + 0 = 18$, $EFT(T_1, R_3) = F_p(T_1, R_3) + EST(T_1, R_3) = 9 + 0 = 9$. Therefore, T_1 is mapped to R_1, $setAvailTime(R_1) = EFT(T_1, R_1) = 6$.

T_5 is the second task to be scheduled in $List_{Priority}$. At this time, $getAvailTime(R_1) = 6$, $getAvailTime(R_2) = 0, getAvailTime(R_3) = 0$. The earliest start times of T_1 on assigned resources are as follows:

$EST(T_5, R_1) = \max\{getAvailTime(R_1), ERT(T_5, R_1)\} = \max\{6, 6 + 0\} = 6$,

$EST(T_5, R_2) = \max\{getAvailTime(R_2), ERT(T_5, R_2)\} = \max\{0, 6 + 19/11\} = 7.727$,

$EST(T_5, R_3) = \max\{getAvailTime(R_3), ERT(T_5, R_3)\} = \max\{0, 6 + 19/28\} = 6.679$,

$EFT(T_5, R_1) = F_p(T_5, R_1) + EST(T_5, R_1) = 13 + 6 = 19$,

$EFT(T_5, R_2) = F_p(T_5, R_2) + EST(T_5, R_2) = 23 + 7.727 = 30.727$,

$EFT(T_5, R_3) = F_p(T_5, R_3) + EST(T_5, R_3) = 15 + 6.679 = 21.679$.

Since $\min_{R_k \in R}\{getAvailTime(R_k)\} = getAvailTime(R_2) = getAvailTime(R_3) = 0$, $\min_{R_n \in R}\{ERT(T_i, R_n) = ERT(T_5, R_1) = 6$, no new resource is considered for T_5. In this case, T_5 is mapped to R_1 as well, $setAvailTime(R_1) = EFT(T_5, R_1) = 19$.

In a similar way, T_7, T_{11} and T_6 are scheduled onto resource R_1, R_3 and R_2, respectively. When T_2 is ready to be scheduled, $getAvailTime(R_1) = 31.00, getAvailTime(R_2) = 36.68$, $getAvailTime(R_3) = 53.36$. The earliest start times of T_2 on currently assigned resources are as follows:

$EST(T_2, R_1) = \max\{getAvailTime(R_1), ERT(T_2, R_1)\} = \max\{31.00, 6 + 0\} = 31$,

$EST(T_2, R_2) = \max\{getAvailTime(R_2), ERT(T_2, R_2)\} = \max\{36.68, 6 + 229/11\} = 36.68$,

$EST(T_2, R_3) = \max\{getAvailTime(R_3), ERT(T_2, R_3)\} = \max\{53.36, 6 + 229/28\} = 53.36$,

$EFT(T_2, R_1) = F_p(T_2, R_1) + EST(T_2, R_1) = 13 + 31 = 44.00$,

$EFT(T_2, R_2) = F_p(T_2, R_2) + EST(T_2, R_2) = 24 + 36.68 = 60.68$,

$EFT(T_2, R_3) = F_p(T_2, R_3) + EST(T_2, R_3) = 17 + 53.36 = 70.36$.

Since $ERT(T_2, R_1) = 6$ is less than the minimum resource available time $getAvailTime(R_1) = 31$, a new resource is considered in this case. As $F_B(C_m, C_m) \gg F_B(C_m, C_n), m \neq n$ in most of cases, we approximate $F_c(i, j, m, n) \approx 0$ in this dissertation for simplicity (such approximation does not affect the proposed approach.)

$EST(T_2, R_1\prime) = \max\{getAvailTime(R_1\prime), ERT(T_2, R_1\prime)\} = \max\{6 + 0\} = 6$,

$EST(T_2, R_2\prime) = \max\{getAvailTime(R_2\prime), ERT(T_2, R_2\prime)\} = \max\{6 + 229/11\} \approx 20.818$,

$EST(T_4, R_3\prime) = \max\{getAvailTime(R_3\prime), ERT(T_2, R_3\prime)\} = \max\{6 + 229/28\} \approx 8.179$,

$EFT(T_2, R_1\prime) = F_p(T_2, R_1\prime) + EST(T_2, R_1\prime) = 13 + 6 = 19.000$,

$EFT(T_2, R_2\prime) = F_p(T_2, R_2\prime) + EST(T_2, R_2\prime) = 24 + 20.818 = 44.818$,

$EFT(T_2, R_3\prime) = F_p(T_2, R_3\prime) + EST(T_2, R_3\prime) = 17 + 8.179 = 25.464$.

As $EFT(T_2, R_1\prime) = 19.000$ is less than $EFT(T_2, R_1) = 44.00$, then a resource R_4 from cluster C_1 is assigned to T_2. The scheduling results of the rest of tasks are shown in Table 5.5 based on the SHEFT algorithm. $\square$

T_i	R_k	$EST(T_i, R_k)$	$EFT(T_i, R_k)$
T_1	R_1	0.000	6.000
T_5	R_1	6.000	19.000
T_7	R_3	19.000	31.000
T_{11}	R_3	31.357	53.357
T_6	R_2	7.727	17.727
T_2	R_2	6.000	19.000
T_4	R_4	9.464	14.464
T_{10}	R_4	27.189	43.189
T_8	R_2	23.887	36.887
T_{12}	R_3	45.003	55.003
T_3	R_5	6.000	22.000
T_9	R_1	22.000	40.000
T_{13}	R_4	53.357	63.357
T_{14}	R_3	63.357	75.357

Table 5.4: The scheduling result for the workflow in Figure 5.2 with the SHEFT algorithm

5.4.3 The SCPOR Algorithm

In the resource selection phase of the SCPOR algorithm, tasks are scheduled according to their orders in $List_{Priority}$, as in the SHEFT algorithm. To minimize the workflow makespan, the SCPOR algorithm is to map all tasks on the *critical path* of the workflow graph to a dedicated resource, so that the data communication costs between tasks on the critical path are eliminated. First of all, we introduce the concept of the critical path as follows.

Definition 5.4.3 (Critical Path CP)**.** Given a workflow graph G, there exists at least one path from T_{entry} to T_{exit}, such that $T_{entry} \rightarrow T_i \rightarrow T_j \rightarrow \cdots \rightarrow T_{exit}$. A set of such a path is denoted as P. A critical path of the workflow is defined as:

$$CP(G) = \{\rho | \rho \in P, \forall T_i \in T_\rho, F_r(T_i) = F_r(T_{entry})\}, \tag{5.15}$$

where a set of tasks in a path $\rho in P$ is denoted as T_ρ. The priority rank of each task in T_ρ is equal to the priority rank of the entry task in the graph. □

If a task T_i's priority rank $F_r(T_i) = F_r(T_{entry})$, then T_i is a *critical task* of the workflow. A set of critical tasks is denoted as $List_{CP}$. A workflow may have multiple critical paths, the SCPOR algorithm chooses one of them and select critical tasks on this path into $List_{CP}$. The procedure to form $List_{CP}$ is implemented by the $getCPList(G, E, R)$ function. As shown in Figure 5.5, $List_{CP}$ initially contains the entry task in the list. From the successors of the entry task, a task T_j is selected into $List_{CP}$ if $F_r(T_j)$ is equal to $F_r(T_{entry})$. Then the successors of T_j will be traversed to select the next task that has the same priority rank with T_{entry}. The procedure is repeated until the search reaches the exit task.

```
     The $getCPList(G, E)$ Function
     Input: $G(T, D, R, F_c, F_{\overline{c}}, F_p, F_{\overline{p}}, F_u, F_d, F_r), E(R_E, C_E, F_M, F_B, F_R)$
     Output: $List_{CP}$
(1)  Begin
(2)      $List_{CP} = \{T_{entry}\}$
(3)      $T_k = T_{entry}$
(4)      while $(T_k \neq T_{exit})$ do
(5)          select $T_j$ where $(T_j \in succ(T_k))$ and $(F_r(T_j) == F_r(T_{entry}))$
(6)          $List_{CP} = List_{CP} \cup \{T_j\}$
(7)          $T_k = T_j$
(8)      end while
(9)  End Function
```

Figure 5.5: The $getCPList(G, E)$ Function

Example 5.4.4. From the Table 5.3, $F_r(T_1) = F_r(T_5) = F_r(T_{11}) = F_r(T_{13}) = F_r(T_{14}) = 117.448$, $List_{CP} = \{T_1, T_5, T_{11}, T_{13}, T_{14}\}$. □

The first step of the SCPOR algorithm (Figure 5.6) is to select a dedicated resource R_C for the execution of all critical tasks. The total computation cost of critical tasks $\sum_{T_i \in List_{CP}, R_k \in R} F_p(T_i, R_k)$ is calculated for each resource, and a resource that produces the minimum cost is selected as R_C (line 3-7). R_C is exclusively used for critical tasks until all of them are scheduled.

In the SCPOR algorithm, the scheduling decision to a task is based on the following four cases: (1) If a task T_i is a critical task ($T_i \in List_{CP}$), then T_i is mapped onto the resource R_C, $< T_i, R_C, EST(T_i, R_C), EFT(T_i, R_C) >$ is added into $List_{Schedule}$, and the available time of R_C is reset to $EFT(T_i, R_C)$ (line 11 - 12). Otherwise, the earliest start time $EST(T_i, R_k)$ and earliest finish time $EFT(T_i, R_k)$ of T_i on a resource R_k is calculated on each of assigned resources except R_C. A resource that produces the smallest earliest finish time ($minEFT$) is assigned to a temporary variable R_S (line 14-18); (2) if T_i is not a critical task and the resource R_S is available by the earliest ready time of T_i, then T_i is mapped to R_S, $< T_i, R_S, EST(T_i, R_S), EFT(T_i, R_S) >$ is added into $List_{Schedule}$, and the available time of R_S is reset to $minEFT$ (line 19-20); (3) if T_i is not a critical task and resource R_S is unavailable at the earliest ready time of T_i, then a new resource R_n will be considered to be assigned to the workflow. If $EFT(T_i, R_S)$ is earlier than $minEFT$, then T_i is mapped to R_n, $< T_i, R_n, EST(T_i, R_n), EFT(T_i, R_n) >$ is added into $List_{Schedule}$. Resource R_n is assigned to R_S and added into R, so R_n can be reused by any other resources after $EFT(T_i, R_S)$ (line 22-25); (4) If T_i is not a critical task and the earliest finish time on any of new resources is later than $minEFT$, then T_i is mapped to R_S, $< T_i, R_S, EST(T_i, R_S), EFT(T_i, R_S) >$ is added into $List_{Schedule}$, and the available time of R_S is reset to $minEFT$ (line 27). The rest of the procedures for checking the idle time of each resource (line 28-31) is similar to the SHEFT algorithm.

Example 5.4.5. For each critical task in Example 5.4.4, (1) if they are scheduled onto R_1, $\sum_{T_i \in List_{CP}} F_p(T_i, R_1) = F_p(T_1, R_1) + F_p(T_5, R_1) + F_p(T_{11}, R_1) + F_p(T_{13}, R_1) + F_p(T_{14}, R_1) = 6 + 13 + 23 + 12 + 15 = 69$; (2) if critical tasks are scheduled onto R_2, $\sum_{T_i \in List_{CP}} F_p(T_i, R_2) = F_p(T_1, R_2) + F_p(T_5, R_2) + F_p(T_{11}, R_2) + F_p(T_{13}, R_2) + F_p(T_{14}, R_2) = 18 + 23 + 26 + 22 + 23 = 112$; (3) if critical tasks are scheduled onto R_3, $\sum_{T_i \in List_{CP}} F_p(T_i, R_3) = F_p(T_1, R_3) + F_p(T_5, R_3) + F_p(T_{11}, R_3) + F_p(T_{13}, R_3) + F_p(T_{14}, R_3) = 9 + 15 + 10 + 10 + 12 = 56$. Therefore, R_3 is selected to be the dedicated resource for all critical tasks. $\square$

According to $List_{Priority}$ in Example 5.4.1, T_1 is the first task to be scheduled, $T_1 \in$

```
    **The SCPOR Algorithm**
    **Input:** $G, E, t_{idle}, List_{Priority}, List_{CP}$
    **Output:** $List_{Schedule}$
(1)**Begin**
(2)    int $minCP = MAX\_NUMBER, R_C = NULL$
(3)    **for** each $R_k \in R$
(4)        $setAvailTime(R_k) = 0$
(5)        **if** $(\sum_{T_i \in List_{CP}} F_p(T_i, R_k) < minCP)$
(6)            **then** $minCP = \sum_{T_i \in List_{CP}} F_p(T_i, R_k), R_C = R_k$
(7)    **end for**
(8)    **while** $(List_{Priority} \neq \emptyset)$
(9)        int $minEFT = MAX\_NUMBER, R_C = NULL$
(10)       select $T_i \in List_{Priority}$ with the highest priority rank
(11)       **if** $(T_i \in List_{CP})$
(12)           **then** map $T_i$ to $R_C$, update $List_{Schedule}$, $setAvailTime(R_C) = EFT(T_i, R_C)$
(13)       **else**
(14)           **for** each $R_k \in (R \setminus R_C)$
(15)               calculate $EST(T_i, R_k)$ and $EFT(T_i, R_k)$
(16)               **if** $(EFT(T_i, R_k) < minEFT)$
(17)                   **then** $minEFT = EFT(T_i, R_k), R_S = R_k$
(18)           **end for**
(19)           **if** $(\min_{R_k \in R}\{getAvailTime(R_k)\} \leq \min_{R_n \in R}\{ERT(T_i, R_n)\})$
(20)               **then** map $T_i$ to $R_S$, update $List_{Schedule}$,
                       $setAvailTime(R_S) = minEFT$
(21)           **else**
(22)               request a new resource $R_n \notin R, \exists R_{n'} \in R, F_M(R_n) == F_M(R_{n'})$
(23)               calculate $EST(T_i, R_n)$ and $EFT(T_i, R_n)$
(24)               **if** $(EFT(T_i, R_n) < minEFT)$
(25)                   **then** map $T_i$ to $R_n$ ,$R_S = R_n, R = R_n \cup R$, update $List_{Schedule}$,
                           $minEFT = EFT(T_i, R_n), setAvailTime(R_n) = minEFT$
(26)               **else**
(27)                   map $T_i$ to $R_S$, update $List_{Schedule}$, $setAvailTime(R_S) = minEFT$
(28)       **for** each $R_k \in (R \setminus R_C)$
(29)           **if** $(minEFT - getAvailTime(R_k)) > t_{idle})$
(30)               **then** $R = R \setminus R_k$
(31)       **end for**
(32)       $List_{Priority} = List_{Priority} \setminus T_i$
(33)   **end while**
(34)   **End Algorithm**
```

Figure 5.6: The SCPOR Algorithm

T_i	R_k	$EST(T_i, R_k)$	$EFT(T_i, R_k)$
T_1	R_3	0.000	9.000
T_5	R_3	9.000	24.000
T_7	R_1	12.393	24.393
T_{11}	R_3	24.750	46.750
T_6	R_2	9.731	19.731
T_2	R_4	9.000	26.000
T_4	R_5	9.000	14.000
T_{10}	R_2	21.231	43.231
T_8	R_4	26.000	38.000
T_{12}	R_4	44.231	54.231
T_3	R_6	12.679	28.679
T_9	R_5	30.393	39.393
T_{13}	R_3	46.750	56.750
T_{14}	R_3	56.750	68.750

Table 5.5: The scheduling result for the workflow in Figure 5.2 with the SCPOR algorithm

$List_{CP}$, so it is scheduled onto R_3. As the entry task, $EST(T_1, R_3) = 0, EFT(T_1, R_3) = F_p(T_1, R_3) + EST(T_1, R_3) = 9, setAvailTime(R_3) = EFT(T_1, R_3) = 9$.

T_5, ranked as the second task in $List_{Priority}$, is also a critical task, so it is scheduled onto R_3 as well. $EST(T_5, R_3) = 9, EFT(T_5, R_3) = F_p(T_5, R_3) + EST(T_5, R_3) = 15 + 9 = 24, setAvailTime(R_3) = EFT(T_5, R_3) = 24$.

The third task to be scheduled in $List_{Priority}$ is T_7. T_7 is not a critical task, so T_7 will not be scheduled on R_3. Since R_1 and R_2 are still available at this time, no new resource is needed to be considered. $EST(T_7, R_1) = \max\{getAvailTime(R_1), ERT(T_7, R_1)\} = \max\{0, \max\{9{+}95/28\}\} = \max\{0, 12.393\} = 12.393$, $EST(T_7, R_2) = \max\{getAvailTime(R_2), ERT(T_7, R_2)\} = \max\{0, \max\{9 + 95/26\}\} = \max\{0, 12.654\} = 12.654$, $EFT(T_7, R_1) = F_p(T_7, R_1){+}EST(T_7, R_1) = 12{+}12.393 = 24.393$, $EFT(T_7, R_2) = F_p(T_7, R_2){+}EST(T_7, R_2) = 26 + 12.654 = 38.654$. Therefore, T_7 is mapped onto resource R_1, $setAvailTime(R_1) = EFT(T_7, R_1) = 24.393$. The scheduling results of the rest of tasks are shown in Table 5.5 based on the SCPOR algorithm.

5.5 Experiments and Discussion

We firstly present the simulation of a services computing environment in Section 5.5.1. Then in Section 5.5.2, our developed workflow generator randomly constructs workflows with various graph attributes. The results of a randomly generated workflow scheduled by the HEFT, SHEFT, CPOP and SCPOP algorithms are analyzed and compared in Section 5.5.3. To further evaluate the performance of our proposed algorithms, extensive experiments on large-scale workflows, compute-intensive workflows, and data-intensive workflows are performed and all statistical experiment results are shown in Section 5.5.4, Section 5.5.5 and Section 5.5.6, respectively.

5.5.1 Computing Environment Simulation

To schedule a workflow, we firstly simulate a computing environment with the following input parameters:

1. The total number of resources in the environment ($|R_E|$). A set of compute resources are created to simulate a computing environment, given the total number of resources $|R_E|$. Each resource has three attributes: a unique resource identifier, a cluster identifier, and the available time of this resource.

2. The total number of clusters in the environment ($|C_E|$). The number of clusters $|C_E|$ can be set between 1 and $|R_E|$.

3. A mapping between resource R_i and C_j ($F_M(R_i) = C_j$). For each resource $R_i \in R_E$, a mapping is randomly generated to assign R_i to a cluster $C_j \in C$. A checking procedure is performed to make sure that there is no empty clusters. ($|C_j| > 0, 1 \leq j \leq |C_E|$).

4. The number of resources assigned to a workflow ($|R|$). A set of resources R from R_E are assigned to a workflow. In our experiments, if $|R| \leq |C_E|$, we randomly select resources from $|R|$ clusters; Otherwise, we firstly select one resource from each cluster, then the rest of resources ($|R| - |C_E|$) are randomly selected from $|C_E|$ clusters.

5. The resource idle time threshold for a workflow (t_{idle}). Once the idle time of a resource $R_i \in R$ is larger than t_{idle}, R_i is considered to be returned from R to R_E.

6. Data transfer rates between cluster C_m and C_n ($F_B(C_m, C_n)$). In our experiments, a data transfer rate between two clusters is randomly generated between 0 and 10 Mbps.

5.5.2 Random Workflow Graph Generation

To evaluate our proposed algorithms, we develop a workflow generator to randomly generate workflow graphs. Each graph is built by the following input parameters:

1. The minimum and maximum numbers of the depth of a graph (MIN_{depth}, MAX_{depth}). In our workflow model, the depth of each workflow has at least two levels (the entry task at the first level and the exit task at the second level), so the minimum depth of the workflow graph is no less than 2 ($MIN_{depth} \geq 2$). The depth of a graph is randomly generated between MIN_{depth} and MAX_{depth}.

2. The minimum and the maximum numbers of vertices at each level ($MIN_{vertex}, MAX_{vertex}$). In our workflow model, only the entry task is at the first level of the graph ($MIN_{vertex} = MAX_{vertex} = 1$), and only the exit task is at the last level ($MIN_{vertex} = MAX_{vertex} = 1$. Other than these two levels, the number of vertices at each level is randomly generated between MIN_{vertex} and MAX_{vertex}. The total number of tasks in a workflow can be calculated by the sum of the vertices at each level.

   Given the above input parameters, the workflow generator randomly generates a workflow graph with the depth between MIN_{depth} and MAX_{depth}, and a number of vertices between MIN_{vertex} and MAX_{vertex} at each level. For each vertex, the connectivity to other vertex is determined by a randomly generated boolean value. If the value is true, an edge is created between these two vertices. After all vertices are processed, a checking procedures are performed to make sure that the out-degree of each vertex in the graph complies with the definition of the workflow.

3. The weight of each vertex ($F_{\overline{p}}(T_i)$). In our workflow model, the average computation cost of a task is represented by the weight of a vertex in the workflow graph. In our experiments, the computation cost of a task on each resource $R_k \in R$ is randomly generated between 0 and 360 hours. A $|T| \times |R|$ matrix of computation costs M_C is constructed for calculating the average computation cost of each task of the workflow.

4. The weight of each edge ($F_{\overline{c}}(i,j)$). In our workflow model, the average data communication cost $F_{\overline{c}}(i,j)$ between task T_i and task T_j is represented by the weight of each edge. In our experiments, the data size transferred between two tasks is randomly generated between 0 and 10 gigabytes.

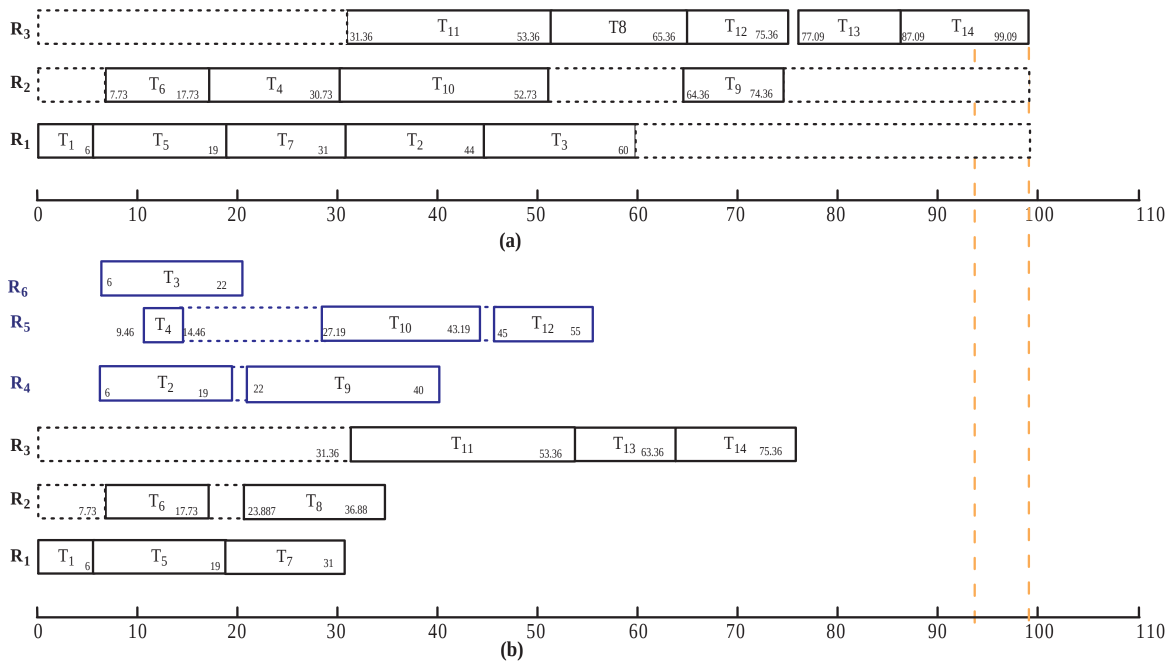


Figure 5.7: The workflow in Figure 5.2 is scheduled by (a) the the HEFT algorithm; (b) the SHEFT algorithm ($t_{idle} = 60$).

5.5.3 Workflow Scheduling Result Analysis

To evaluate our proposed algorithms, we develop the HEFT, CPOP, SHEFT and SCPOR algorithms and apply them to schedule the workflow in Figure 5.2, which is randomly generated by our developed framework. Three resources, R_1, R_2 and R_3, are initially assigned to the workflow from the simulated computing environment. They are in clusters C_1, C_2 and C_3, respectively. The average computation cost of each task in the workflow are given in Table 5.2, and the average data communication cost can be calculated given the information in Table 5.1.

In Figure 5.7, the results scheduled by the HEFT and the SHEFT algorithms are shown in the two Gantt charts. The workflow makespan scheduled by the SHEFT algorithm is 75.36, 24% percent improved from the makespan scheduled by the HEFT algorithm (99.09). The vertical dashed line in the Figure 5.7 shows the gap of workflow makespan scheduled by the two algorithms. Using the HEFT algorithm, R_1, R_2 and R_3 are consumed by this workflow until the exit task T_{14} completes. The total usage of the three resources is 297.28 seconds; while the resources can be dynamically changed in SHEFT algorithm, the total resource usage is 246.92 seconds, 17% improved from the HEFT algorithm.

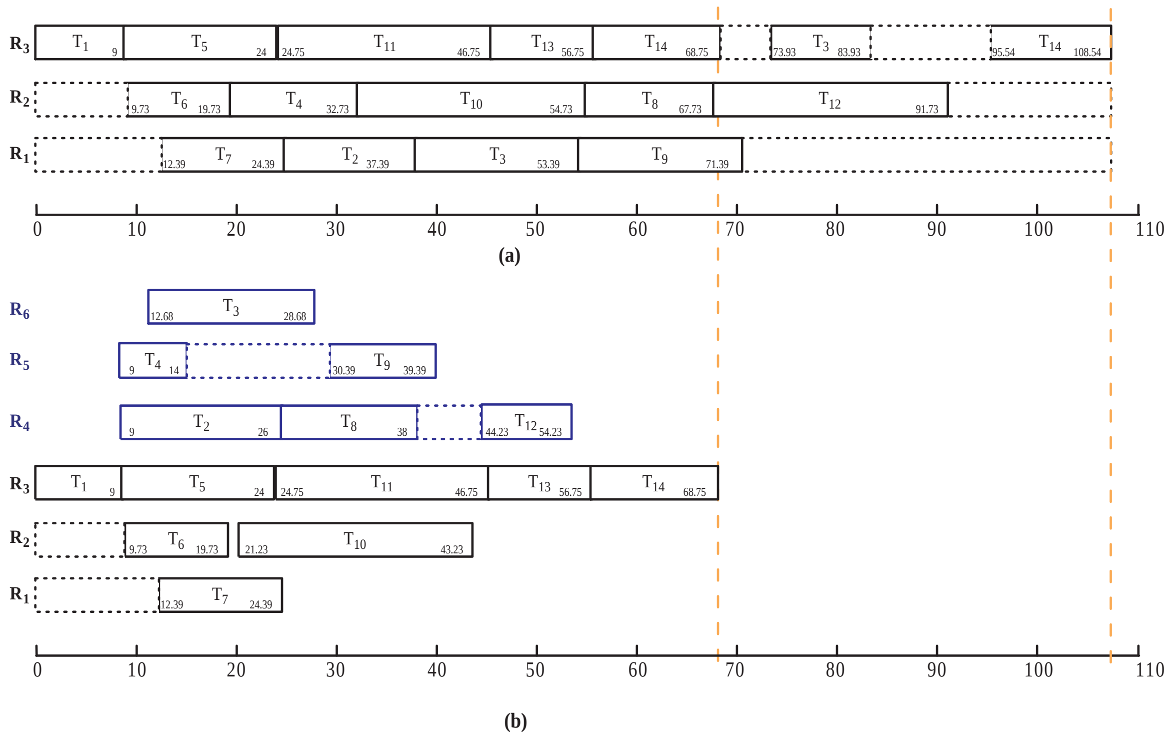


Figure 5.8: The workflow in Figure 5.2 is scheduled by (a) the the CPOP algorithm and (b) the SCPOR algorithm ($t_{idle} = 20$).

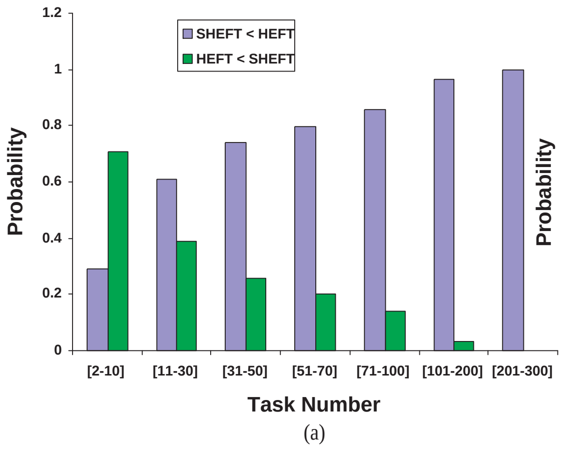
In Figure 5.7.(b), T_1, T_5 and T_7 are scheduled on R_1, T_6 and T_{11} are scheduled on R_2 and R_3. When T_2 is ready to be scheduled, all the three assigned resources are unavailable until T_7 completes at the 31st second. In this case, a new compute resource $R_4 \in C_1$ is assigned to T_2. R_4 is available for other tasks after T_2 completes at the 19th second. Then T_4, T_{10}, T_{12} are assigned to new resource $R_5 \in C_3$, T_3 is assigned new resource $R6 \in C_1$. Because of the introduced three resources R_4,R_5 and R_6 , $T_2.T_3, T_4, T_9, T_{10}, T_{12}$ complete earlier than the time they complete using the HEFT algorithm.

During the workflow execution, the number of resources is equal to 3 at the beginning (R_1, R_2 and R_3), then increases to 5 at 6th second ($R = \{R_1, R_2, R_3, R_4, R_5\}$), and further increases to 65 at 9.46th second ($R = \{R_1, R_2, R_3, R_4, R_5, R_6\}$). R_6 is released at the 22nd second, followed by R_1 released at 31st second, R_2 released at 36.88th second, and R_4 released at 40th second, R_5 released at 55th second. R_3 is finally released to the computing environment at the 75.36th second. The horizontal dash lines indicate the idle time of the resources.

The two Gantt charts in Figure 5.8 has shown that the SCPOR algorithm performs much better than the CPOP algorithm in terms of the workflow makespan and resource usage. More specifically, the workflow makespan scheduled by the SCPOR algorithm is 68.75, 36.66% percent improved from the makespan scheduled by the CPOP algorithm (108.54). The resource usage scheduled by these two algorithms are 325.62 and 227.99, respectively. The resource usage by the SCPOR algorithm is 30% less than the CPOP algorithm.

5.5.4 Statistical Performance Evaluation on Scalable Workflows

To evaluate the scalability performance of our proposed algorithms, we firstly set 7 ranges for the number of tasks in a workflow. That is $[2, 10]$, $[10, 30]$, $[31, 50]$, $[51, 70]$, $[71, 100]$, $[100, 200]$ and $[200, 300]$. For each range, our developed workflow generator randomly generates 50,000 workflow graphs. The communication costs and computation costs of these workflows are also randomly generated within defined reasonable ranges. Then we schedule these workflows by the HEFT, SHEFT, CPOP and SCPOR algorithms and compare their scheduled workflow makespan as the size of the workflows increases. The total number that one algorithm outperforms another is counted, divided by the total number of experiments $(50,000)$ is considered as the probability of this algorithm outperforms the other algorithm. For example, in Figure 5.9.(a), there are $14,550$ out of $50,000$ times that the workflow makespan scheduled by the SHEFT is less than that scheduled by the HEFT algorithm, when the number of tasks of a workflow is less than 10. Therefore, the probability that SHEFT outperforms HEFT is $14550/50000 = 29.1\%$, and the probability that HEFT outperforms SHEFT is $1 - 29.1\% = 70.9\%$.



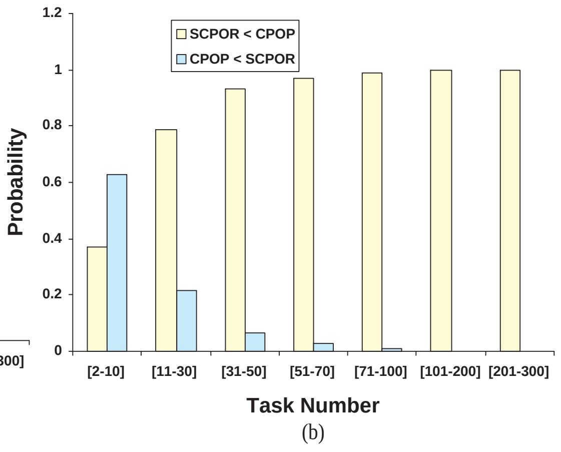


Figure 5.9: Scheduling 50,000 randomly generated workflows for the number of tasks in each range. The comparison of scheduling results (a) between SHEFT and HEFT and (b) between SCPOR and CPOP.

Although the HEFT algorithm shows better performance when the number of tasks is less than 10, as the size of workflows increases, our proposed algorithm exhibits better performance as follows: When tasks of a workflow increases to the range of $[11, 30]$, the probability of SHEFT outperforms HEFT goes up to 61.1%, then the probability continues to increase from 74.0%, 79.7%, 85.9% to 96.7%, as the number of tasks in a workflows is increased from $[31, 50]$, $[51, 70]$, $[71, 100]$ to $[100, 200]$. When the number of tasks in a workflow is more than 200, all $50,000$ workflows scheduled by SHEFT outperform better than these workflows scheduled by HEFT.

The performance between the CPOP and SCPOR algorithm follows the same trend. When the number of tasks is in the range of $[2, 10]$, the probability of CPOP outperforms SCPOR is 63% while the probability of SCPOR wins CPOP is 37%. However, as the number of tasks increases, the performance of SHEFT is shown better and better. As shown in Figure 5.9.(b), the probability keeps rising from 78.6%, 93.3%, 97.2% to 98.9%, as the number of tasks in a workflows increases from $[31, 50]$, $[51, 70]$, $[71, 100]$ to $[100, 200]$. When the number of tasks reaches to the range of $[101, 200]$, the probability that SCPOR outperforms CPOP becomes 1, it means that none of the scheduling results in the $50,000$ experiments scheduled by CPOP is better than the results scheduled by SCPOR. When the number of tasks continue to increase to $[201, 300]$, it is still shown that SCPOR outperforms CPOP in all another $50,000$ experiments.

5.5.5 Statistical Performance Evaluation on Compute-intensive Workflows

To investigate the performance for compute-intensive workflows, our developed workflow generator randomly generates $50,000$ workflow graphs for the number of tasks in the ranges of $[2, 50]$, $[51, 100]$, $[101, 150]$, $[151, 200]$ and $[201, 250]$. For each workflow, the total number of computation costs of tasks is at least 50 times greater than the communication costs. we schedule these workflows by the HEFT, SHEFT, CPOP and SCPOR algorithms and compare their scheduled workflow makespans with different ranges of task numbers. As shown in Figure 5.10.(a), when the number of tasks are in the range of $[2, 50]$, the probability of SHEFT outperforms HEFT is 90.2% while the probability of HEFT outperforms SHEFT is 9.8%. From the number of tasks in the range of $[50, 100]$, all the scheduling results scheduled by SHEFT are better than the results scheduled by HEFT. In Figure 5.10.(b), all scheduling results scheduled by SCPOR outperforms the results scheduled by CPOP for the number of tasks at all ranges. In this case, all the experiments have shown that our proposed algorithms outperforms the HEFT and CPOP algorithms for compute-intensive workflows.

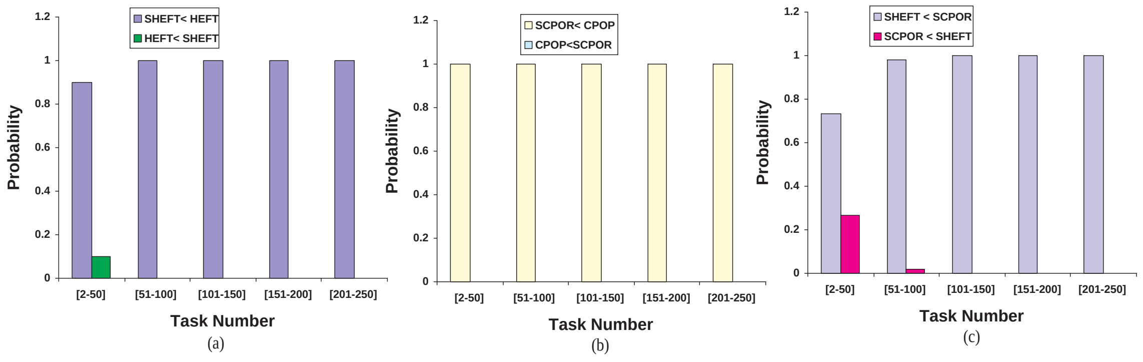


Figure 5.10: Scheduling 50,000 randomly generated compute-intensive workflows. The comparison of scheduling results (a) between SHEFT and HEFT, (b) between SCPOR and CPOP, and (c) between SHEFT and SCPOR.

When the number of tasks are in the range of $[2, 50]$ and $[51, 100]$, the probability of SHEFT outperforms SCPOR is 73.2% and 98%, as shown in Figure 5.10.(c). When the task number continues to grow, all the $50,000$ experiments for each range of task number have shown that SHEFT outperforms SCPOR. It means that SHEFT wins the best for scheduling large-scale and compute-intensive workflows that are mostly common in scientific computing.

5.5.6 Statistical Performance Evaluation on Data-intensive Workflows

To investigate the performance for data-intensive workflows, our developed workflow generator randomly generates $50,000$ workflow graphs for the number of tasks in the ranges of $[2, 50]$, $[51, 100]$, $[101, 150]$, $[151, 200]$ and $[201, 250]$. For each workflow, the total communication cost between tasks is at least 50 times greater than the communication cost of the workflow.

Although the advantage of the SHEFT algorithm over the HEFT algorithm is not as distinct as in compute-intensive workflows, the probability of SHEFT over HEFT is shown a consistent increase as the size of workflows grows. As shown in Figure 5.11.(a), the probability of SHEFT outperforms HEFT is 69.4% with the number of tasks in the range of $[2, 50]$, then it keeps rising from 76.7% to $80.8\%, 82.9\%, 83.3\%$, when the number of tasks from the range of $[51, 100]$ to $[101, 150]$, $[151, 200]$, $[201, 250]$. Similar trend goes even faster between the SCPOR algorithm

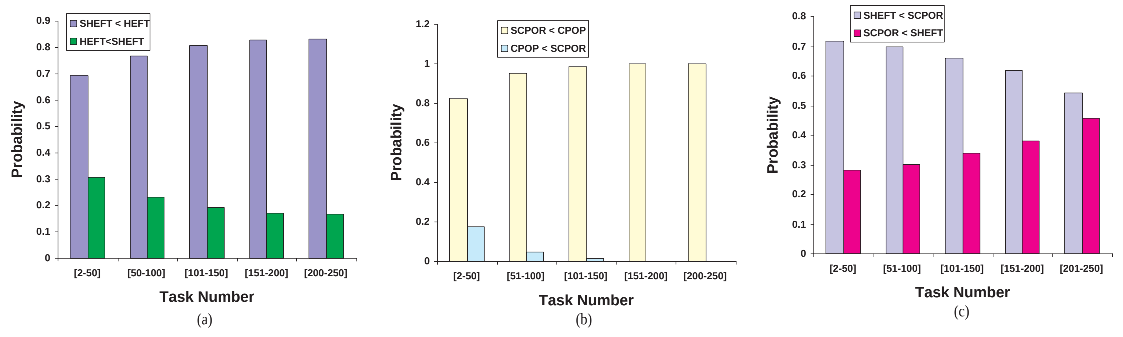


Figure 5.11: Scheduling 50,000 randomly generated data-intensive workflows. The comparison of scheduling results between (a) SHEFT and HEFT, between (b) SCPOR and CPOP, and between (c) SHEFT and SCPOR.

and the CPOP algorithm. As shown in Figure 5.11.(b), the probability that SCPOR outperforms CPOP continuously increases from $82.3\%, 95.2\%, 98.5\%, 99.8\%$ to 1, when the number of tasks from the range of $[2, 50], [51, 100], [101, 150], [151, 200]$ to $[201, 250]$.

In Figure 5.11.(c), the probability of SCPOR larger than SHEFT increases from 28.4% in the range of $[2, 50]$ to 30.3% in the range of $[51, 100]$. The probability increases continuously from 30.3% to $34.1\%, 38\%$ and to 45.8%, as the number of tasks increases from the range of $[51, 100]$ to $[201, 250]$. After that, the workflow makespans scheduled by these algorithms are very close. It is because that as the size of workflows grows, the data communications between critical tasks increase as well, the advantage of assigning all critical tasks onto one resource may become more significant for minimizing workflow makespan. However, as the number of tasks continues to grow to over 250, the data communications between tasks that are not on the the critical path become more dominant, workflow scheduling results by the proposed algorithms are shown statistically very close to each other.

CHAPTER 6: THE VIEW SCIENTIFIC WORKFLOW MANAGEMENT SYSTEM

In order to validate the feasibility of our proposed reference architecture, we propose a service-oriented architecture for our developed VIEW system that complies with the reference architecture. In this chapter, we firstly present our architectural design principles that serve the foundation for the design of the VIEW system in Section 6.1; these principles are desirable requirements from a general software engineering perspective rather than requirements specifically essential for SWFMSs. Secondly, we introduce the overall VIEW system architecture and the architectures of subsystems in Section 6.2. Then, we present a service configuration management in the VIEW system that provides a flexible configuration functionality for the VIEW Kernel and VIEW Task Executors, in Section 6.3 and Section 6.4. In Section 6.5, we summarize the advantages of using SOA in SWFMS development. Finally, our developed VIEW based workflow application called FiberFlow system is presented in Section 6.6.

6.1 Architectural Design Principles

In addition to the principles described in [9], the development of the VIEW system comply with the following principles to satisfy the requirements of SWFMSs:

P1: Loose-coupling. In the VIEW system, each subsystem is a loosely-coupled, autonomous, reusable, and discoverable service component, and each service component communicates with others by simply requesting their services with the interfaces described by WSDL. Changing the implementation of one service component while remaining the same interfaces does not affect other service components. Service components interact with each other by Web service invocation using SOAP messages via Internet-based protocols.

P2: Localized database access. In the VIEW system, a service component is not allowed to directly access the databases that are managed by other service components; instead, a service component accesses data by requesting services provided by other service components. There are two reasons behind this: 1) databases for each service component are configurable, which provides a more flexible implementation for each subsystem on demand, so different service components are allowed to share the same database or each service component can use their own databases; and 2) the design of models and data management for each service component may change, but such changes can be transparent to other service components by using the same interfaces.

P3: Model-based service component. The granularity of services, i.e., how fine or coarse grained services should be designed, is an important issue for system development. In the VIEW system, the granularity of services is based on the granularity of data models, that is, all operations over one data model is grouped into one service and described by one WSDL, while operations over different data models are separated into different services. In this way, the modification of one data model (e.g., a task model) will not affect the functionality of another data model (e.g. a task run model).

6.2 Overall Architecture and Subsystem Architectures

The overall architecture of VIEW in Figure 6.1 consists of six service components that correspond to the main functional subsystems proposed in the reference architecture. Other than *Workbench*, the interface for each service component is defined and described by WSDL: $I_{WE}, I_{WM}, I_{TM}, I_{PM}$ and I_{DPM} for the interface of the Workflow Engine, the Workflow Monitor, the Task Manager, the Provenance Manager, and the Data Product Manager, respectively, which comprises the VIEW *Kernel*. In the following, we focus our discussion on the architectural details of the VIEW Kernel.

Workbench. The Workbench subsystem implements the functions of workflow design, presentation, and visualization identified at the Presentation Layer in the reference architecture. Currently it consists of five components (see Figure 6.2 (left)): *Workflow Designer*, *Provenance Explorer*, and the *GUIs* for the VIEW Kernel.

Workflow Designer provides a scientist-friendly GUI for the design and modification of

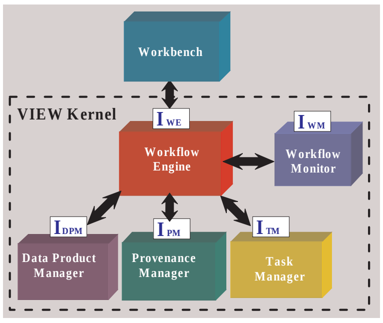


Figure 6.1: Overall architecture of the VIEW system.

scientific workflows. A scientist can drag and drop registered tasks and data products into the design panel and link them one to another using various dataflow and controlflow constructs. Workflow Designer is supported by our proposed workflow specification language, called *Scientific Workflow Language* (SWL) to define a scientific workflow, according to the VIEW *Workflow Model*, which supports hierarchical (nested) scientific workflows. Workflow definitions in the Workflow Designer are saved in XML files into a *Local Workflow Repository*. A workflow definition in the VIEW Workbench consists of three parts: 1) a *workflow specification* to store the logical structure and its constituent components; 2) *workflow run parameters* to store all parameters for each task run; and 3) a *workflow layout* to store the graphical layout of the scientific workflow that is required to display the workflow in the Workflow Design Panel. The first two parts are needed for the execution of a workflow run, and the last part is to display and manipulate a scientific workflow in the VIEW Design Panel.

Provenance Explorer enables a user to browse and visualize scientific workflow provenance metadata. Moreover, together with the GUI for the *Data Product Manager*, one can present and visualize various data products, from simple data values and plain texts to complex data types.

The VIEW Workbench supports Windows-based user interfaces for the VIEW Kernel, while reusing the same service components. These scientist-friendly GUIs interact with subsystems via I_{TM}, I_{WM}, I_{PM}, and I_{DPM}, respectively. This leads to the architectural flexibility to allow scientists to customize their own GUIs for each particular SWFAS, thus satisfying requirement R1.

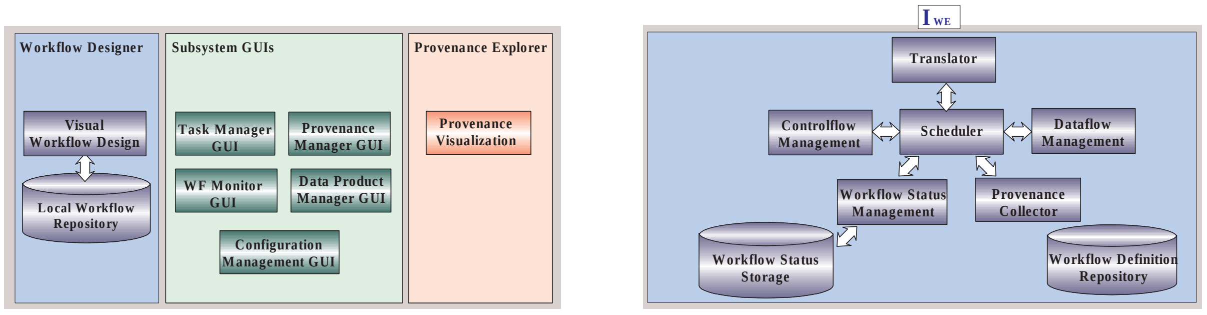


Figure 6.2: Architecture of the VIEW Workbench (left) and the VIEW Workflow Engine (right).

Workflow Engine. The architecture of the Workflow Engine subsystem is shown in Figure 6.2 (right). Centered around *Scheduler*, the *Workflow Engine* consists of six functional modules: *Scheduler*, *Translator*, *Controlflow Management*, *Dataflow Management*, *Workflow Status Management*, and *Provenance Collector*.

First, Translator provides a mapping scheme for translating a workflow specification into an optimized internal executable workflow representation. Workflow definitions delivered from Workflow Designer are saved into the *Workflow Definition Repository* via I_{WE}. A workflow definition in Workbench's Workflow Repository should be consistent with the version in Workflow Definition Repository during workflow execution. Second, the separation of controlflow and dataflow management from workflow scheduling greatly improves the extensibility of the VIEW Workflow Model since the introduction of additional controlflow or dataflow constructs can be achieved by upgrading their individual modules without modifying other modules. Third, as Scheduler is able to support multi-thread processing, it can initialize and maintain a number of workflow runs simultaneously, *Workflow Status Storage* provides a foundation for workfow run monitoring and failure handling (requirement R6). Finally, Provenance Collector is responsible for collecting all provenance information and storing them into Provenance Manager via I_{PM}. Since the VIEW Workflow Engine supports an open and extensible scientific workflow language and is loosely coupled with other subsystems, the workflows/subworkflows designed by other SWFMSs can directly request to and invoked by the VIEW Workflow Engine via the Web service communication and invocation. Thus, the sharing and mapping between the VIEW Workflow Engine and other SWFMSs can be greatly facilitated (requirement R7: level 2).

In contrast to BWFMSs that mostly manage controlflow oriented workflows, in which the order of task execution is explicitly specified by controlflow constructs, such as sequential, conditional, and loop, the VIEW Workflow Engine is developed for dataflow-driven scientific workflows. As a result, the availability of input data for a task initiates its execution, and the movement of data via data channels determines the execution order of a workflow.

Workflow Monitor. Our current implementation of the Workflow Monitor uses a Publish/Subscribe model [114] and focuses on the implementation of monitoring workflow execution status. Future implementation will introduce other features including forward and backward recovery in the case of failures.

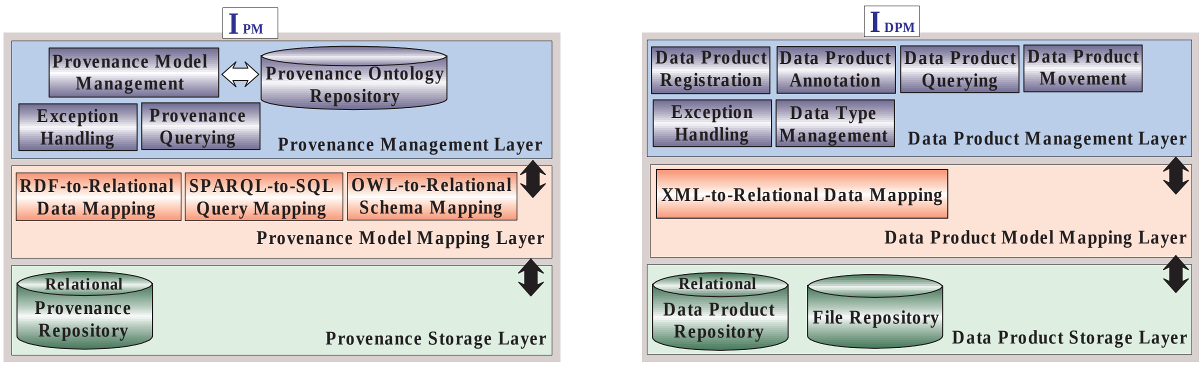

I_{PM}
Provenance Model Management
Provenance Ontology Repository
Exception Handling
Provenance Querying
Provenance Management Layer
RDF-to-Relational Data Mapping
SPARQL-to-SQL Query Mapping
OWL-to-Relational Schema Mapping
Provenance Model Mapping Layer
Relational Provenance Repository
Provenance Storage Layer
I_{DPM}
Data Product Registration
Data Product Annotation
Data Product Querying
Data Product Movement
Exception Handling
Data Type Management
Data Product Management Layer
XML-to-Relational Data Mapping
Data Product Model Mapping Layer
Relational Data Product Repository
File Repository
Data Product Storage Layer

Figure 6.3: Architectures of the VIEW Provenance Manager (left) and the VIEW Data Product Manager (right).

Provenance Manager. The architecture of the Provenance Manager subsystem shown in Figure 6.3 (left) includes three layers: the *provenance management* layer, the *provenance model mapping* layer and the *provenance storage* layer.

The provenance management layer is responsible for the representation of workflow run provenance via domain ontologies that serve as vocabularies to describe and serialize provenance metadata. It consists of two modules: *provenance model management* and *provenance querying*. Provenance Model Management manages the ontologies that include both general provenance vocabularies and domain-specific ontologies used to represent knowledge in a particular scientific field, e.g., Biology or Physics (requirement R2). To address the requirements of provenance representation interoperability, extensibility, and semantic integration in VIEW, we use Semantic Web technologies for provenance representation. In particular, Web Ontology Language (OWL) is used to express ontologies, and Resource Description Framework (RDF) is used to serialize provenance metadata. Provenance querying [115] is expressed by RDF query language SPARQL. Exception Handling analyzes all errors reported and implements several strategies to resolve them, so the subsystem can continue functioning.

The provenance model mapping layer serves as an integration medium between the provenance management layer and the provenance storage layer. It currently contains three mappings: 1) *OWL-to-Relational schema mapping* to generate a relational database schema based on an ontology that is used to represent provenance metadata; 2) *RDF-to-Relational data mapping* to map provenance metadata in RDF to relational tuples and store them into the relational database, and 3) *SPARQL-to-SQL query mapping* to translate provenance queries in SPARQL into relational queries in Structured Query Language (SQL) that can be executed by the RDBMS. The main challenge of this layer is to provide various efficient semantics-preserving mappings between different data models. More details on provenance storage and querying in VIEW are available in [116, 115], where a sample provenance ontology is described and the three mappings are further studied and experimentally evaluated.

The relational model layer includes a relational provenance storage implemented using a Relational Database Management System (RDBMS), which serves as an efficient backend to store and query provenance metadata. In this layer, provenance metadata is stored in relational tables and queried using SQL. The requirements addressed by this layer include efficiency and scalability of provenance metadata management.

Data Product Manager. The architecture of Data Product Manager subsystem shown in Figure 6.3 (right) consists of three layers: the *data product management* layer, the *data product model mapping* layer, and the *data product storage* layer.

The data product management layer consists of a set of modules that are responsible for the management of data products based on the VIEW *Data Product Model*. The Data Product Manager allows scientists to access various data products transparently with respect to their heterogeneity and distribution (requirement R4), supported by *Data Product Registration*, *Annotation*, and *Querying*. The *Data Type Management* module defines and manages all required data types to support data storage and task execution. All VIEW subsystems use the same set of data types that are defined by the Data Product Manager, so the introduction of a new data type in *Data Type Management* becomes effective to all other subsystems. The *Data Product Movement* mainly has three functions: 1) data products are allowed to be moved from client-side to *Data Product Repository* or *File Repository* during data product registration via I_{DPM}; 2) data products sometimes have to be moved from where they are registered to where a task resides in order to execute the task; and 3) data products produced by workflow execution can be moved back to Data Product Repository/File Repository, or registered with the Data Product Manager via I_{DPM}.

The data product model mapping layer serves as an integration medium between the data product management layer and the data product storage layer. The rationale for such an architecture is that for different scientific domains, there are different data product models, implementations, and storage approaches that may require different mapping schema, but sharing the same architecture. One can introduce new data product models, new implementations, or new storage approaches by simply adding new mapping modules in this layer, without affecting modules in other layers.

In the data product storage layer, the Data Product Manager employs a relational Data Product Repository to store data products metadata, and File Repository to store files, so *XML-*

to-Relational data mapping [117, 118] is required in data product model mapping layer to map XML-modeled data products into relational databases.

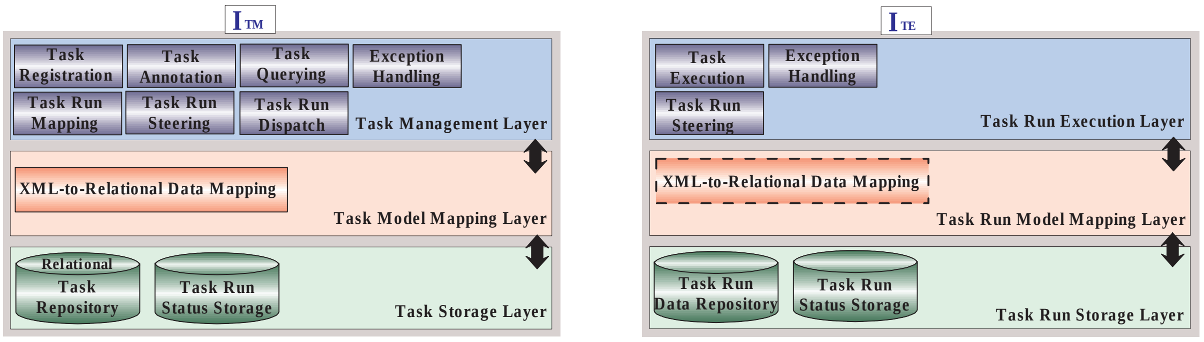


Figure 6.4: Architectures of the Task Master (left) and the Task Executor (right).

Task Manager. The VIEW Task Manager supports a distributed architecture, consisting of a Task Master and a set of Task Executors of various types.

The architecture of the Task Master shown in Figure 6.4 (left) consists of three layers: the *task management* layer, the *task model mapping* layer, and the *task storage* layer.

The task management layer provides a set of modules that are responsible for the management of tasks and task runs, based on the VIEW *Task Template Model* and the VIEW *Task Run Model*. The Task Master allows scientists to register/delete a task transparently with respect to its heterogeneity and distribution (requirement R3), which is supported by the modules of *Task Registration, Annotation, and Querying*. The task run execution and management are handled by the models of *Task Run Steering, Mapping, and Dispatch*. More specifically, Task Run Steering is used to listen to requests from other subsystems to create/abort/pause/resume a task run via I_{TM}. Task Mapping performs a dynamic mapping from an abstracted task interface to a task component containing a physical implementation, and then delivers the task run to *Task Run Dispatch*, where a Task Executor is dynamically assigned to execute the task component.

In the task model mapping layer, *XML-to-Relational data mapping* [117, 118] is required to map XML-modeled task template specifications into relational *Task Repository*, and map XML-modeled task run descriptors into the relational *Task Status Storage*. The extensions of various mapping mechanisms are allowed to plug-in the task model mapping layer to incorporate heterogeneous data-model storages in the task storage layer.

In order to support the distributed execution of tasks in a wide range of heterogeneous environments (requirement R3), a new architectural subsystem called Task Executor is introduced. Task Executor improves the VIEW system in the following aspects: 1) it separates the task and task run management environment in Task Master from the task execution environment in Task Executors; 2) task execution becomes more reliable as tasks can be executed on distributed Task Executors, avoiding the problem of a centralized architecture that may suffer from a single point of failures; 3) different tasks for one workflow can be executed in parallel at distributed nodes to improve performance and efficiency; and 4) the integration of a new type of service or application is achieved by the extension of one Task Executor, without affecting other Task Executors and the Task Master.

The architecture of the Task Executor consists of three layers: the *task run execution* layer, the *task run model mapping* layer, and the *task run storage* layer.

The task run execution layer provides a set of modules that control the task run execution based on the VIEW Task Run Model. *Task Run Steering* is used to listen to requests from Task Master to abort/pause/resume a task run via I_{TM}. *Task Execution* performs data movement and invokes a task component for a task run.

The implementations at the task run model mapping layer and the task run storage layer vary from the following scenarios: task run status can be simply maintained in the main memory, instead of a persistent storage. However, for large-scale workflows that are composed of a great number of tasks, maintaining a large number of task run status has to rely on a persistent *Task Run Status Storage*, such as a relational database. Then *XML-to-Relational data mapping* [117, 118] is required at the task run model mapping layer to map XML-modeled task run descriptors into relational *Task Run Status Storage*. *Task Run Data Repository* at the task run storage layer is used to save temporary data products that are moved from distributed data sources for a task execution.

Some SWFMSs are built upon a monolithic system or a centralized database that acts as a single point of failure: when a component of the system or a database fails, there is no way to continue executing workflows. In response to this issue, the VIEW system is composed of a set of loosely-coupled and distributed service components, and each service component has multiple alternative services distributed on other machines. Accordingly, there is a need for the management of these services to provide higher system availability.

6.3 VIEW Kernel Configuration Management

The VIEW Kernel consists of several loosely-coupled service components, and each of them could have multiple backup services deployed on different machines. One service component may have various implementations, but sharing the same WSDL. All of these service components are deployed at distributed environments.

In addition, a database in the VIEW system could specifically serve one service component, or multiple backup service components, or even be shared by serval service components of the VIEW Kernel. To support database failover, one serving database could also setup several mirror databases on distributed machines, and each of them is kept synchronized with the serving database, so that once this database fails, its service component(s) can switch to any other mirror databases.

To enable an on-demand VIEW Kernel, the VIEW system provides a service component, called VIEW *Configuration Management*, to manage the configurations of all VIEW Kernel service components and their serving databases. First, the Configuration Management GUI embedded in the VIEW Workbench allows scientists to register all deployed service components for the VIEW Kernel and their serving database(s). The service components for a VIEW Kernel subsystem employ the same WSDL to describe their common interfaces. Second, when the VIEW system is adopted by a specific SWFAS, a template of the VIEW system can be composed on demand by configuring each VIEW Kernel service component and its database(s), which are already registered in the Configuration Management. Third, such template of the system can invoke the chosen services during the runtime. Once a service of the template is unavailable, configuration management will invoke another alternative service. As the alternative service and unavailable service share the same database(s) and repositories in their subsystem storage layer, workflow run and task run status are still valid, which makes it possible to resume the workflow execution starting from the service downtime.

6.4 Task Executor Configuration Management

To support the invocation and execution of various heterogeneous task components, the VIEW Task Manager introduces several types of Task Executors, with each type of Task Executor corresponding to a type of tasks with regard to their programming languages, invocation mechanisms and computing environments. Each type of Task Executors shares one common implementation that is different from other types. Each Task Executor can be deployed either at the host of the Task Master or at any other standalone host. All Task Executors employ the same architecture and the same WSDL.

To improve the failover in task execution, the VIEW system provides Task Executor Configuration Management to manage the configuration of all Task Executors. First, the Configuration Management GUI embedded in the VIEW Task Manager allows scientists to register all available Task Executors. Second, an appropriate Task Executor can be automatically assigned to a task by the VIEW Task Master during task execution. Finally, if a Task Executor happens to be unavailable during task runtime, an alternative Task Executor will be chosen to retry the execution. In this case, the whole scientific workflow does not need to be aborted or restarted, as other Task Executors will not be disturbed during the failover procedure of the failed Task Executor.

6.5 Advantages of Using SOA in SWFMSs

While the emergence of SOA as an architectural paradigm provides many benefits for distributed computing [119], we identify the following advantages of using SOA specifically for the development of an SWFMS:

1) *Service loose coupling*. Service loose coupling minimizes the dependencies among subsystems of an SWFMS by the definitions of a set of language and platform independent interfaces. In our proposed architecture, each subsystem's functionality is exposed as a Web service. As a result, an SWFMS can be composed on demand from various subsystems provided by different parties as Web services. One can also easily switch from one service to another for each subsystem. For example, there may be several provenance management services available, and using SOA, one can use and switch any provenance management service on demand for a specific SWFMS.

2) *Service abstraction and autonomy*. A Web service provides an abstract interface that is independent from its implementation. In addition, each Web service is autonomous in the sense that a service provider has the control over the application logic that the Web service encapsulates. As a result, a service provider can dynamically change the implementation and deployment environment of a Web service for a subsystem of an SWFMS with no downtime for the SWFMS as long as such changes do not affect the defined interface. Such autonomy also greatly facilitates the management of the development and evolution of the whole system.

3) *Service reusability*. As each subsystem of an SWFMS becomes a uniform computing unit with standard interface descriptions and universal accessibility through standard communication protocols, it can be reused across various SWFMSs, even simultaneously used by both local SWFMSs and other SWFMSs across the Internet.

4) *Service discoverability*. As each subsystem of an SWFMS is implemented as a Web service that is enriched with a semantic description, one can register the service in some public service registries. As a result, a subsystem becomes discoverable and can be selected and used

by other SWFMSs on demand.

5) *Service interoperability.* Service interoperability is enabled by the open standards of messages and communication protocols for Web services, which are supported by a large body of IT industry and the Web Services Interoperability Organization (WS-I). Using Web services, the interoperability across various SWFMSs (requirement R7: level 3) can be greatly improved.

6.6 VIEW based FiberFlow System

VIEW is an *application-independent* SWFMS, serving as a foundation on which various SWFASs can be developed according to their own domain-specific requirements. The FiberFlow system is such an SWFAS developed for automatic transforming the large-scale neuroimaging data to knowledge through cross-subject, cross-modality computation, ultimately leading to high clinical intelligence and more informed and accurate decision making in various neural diseases.

The complexity of workflows in the FiberFlow system poses the grand challenges which can be summarized in the following three aspects: First, each workflow may produce a large amount of processed data under different experimental parameter settings. All analytical results and intermediate results vary from data types and formats, and therefore generate great challenges for data management. Second, some computation-intensive tasks are required to be performed on distributed Grid environments in order to reduce the wait time. Third, most tasks in FiberFlow are also interaction-intensive and visualization-intensive, as domain scientists need to frequently manipulate, process, and evaluate the imaging data. Due to the aforementioned challenges, an SWFMS is ideal to manage all the research artifacts to speed up the effective FiberFlow exploratory process.

Figure 6.5 (right) demonstrates a typical workflow designed in the FiberFlow system which is to automate population-based statistical analysis of the variances of fiber bundle shapes and fiber connectivity strengths among normal individuals and patients. Most data products involved in this workflow are volume image files, such as Magnetic Resonance Imaging (MRI) and Diffusion Tensor Imaging (DTI). They are stored in the *Analyze* format, which is one of the standard formats for 3D medical imaging. All tasks in this workflow are implemented as Windows-based applications, and some of them require client-side user interactions to identify *Regions of Interests* (ROIs).

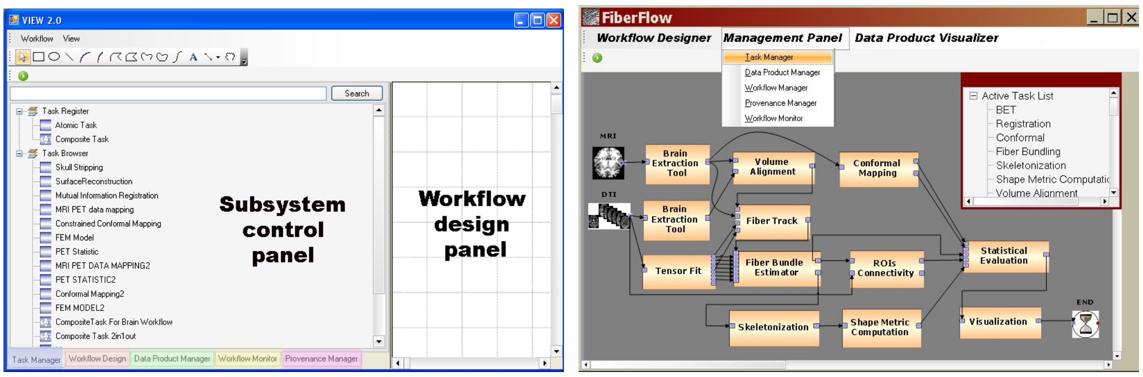


Figure 6.5: The VIEW Workbench embedded with five subsystem control panels and a workflow design panel (left), and a user-interaction intensive and visualization intensive scientific workflow displayed in a customized GUI for the view based FiberFlow system (right).

In a nutshell, the first task is to perform segmentation using the *Brain Extraction Tool* (BET), which segments a subject's neocortex based on DTI and MRI imaging data, and outputs new volume image files with the subject's skull being stripped away. Those files become the input for the *Volume Alignment* (VA) task. This task conducts spatial mappings between the two skull-stripped DTI and MRI files, and generates a text file containing a matrix of all mapping parameters. Besides BET, another preprocessing task is called *Tensor Fit* (TF). It computes tensor fields using DTI data, and generates various invariant metrics which, together with other outputs derived from BET and VA, are inputs of the *Fiber Tracking* task for fiber tractography [120]. To derive a population-based statistics on human brains of varying sizes and shapes, the *Conformal Mapping* (CM) task is applied to perform an inter-subject registration, which maps skull-stripped MRI to a common 3D template space. Meanwhile, the text-based output from the Fiber Track task, together with all volume files from TF, are applied with the Fiber Bundle Estimator task to identify the specific fiber bundle of interest and computes its isosurface. The user interactions for picking up ROIs are required in this task. Then a volume file recording ROI bundles is produced and supplied to the *ROIs Connectivity* (RC) task. Based on the DTI imaging, RC creates a probabilistic model based on the Bayesian inference theory, and estimates connectivity between ROIs. The task's output recording 3D fiber bundles becomes the input of *Skeletonization*, and the result from Skeletonization is supplied to the *Shape Metric Computation* (SMC) task to generate quantitative shape descriptors. Finally, the *Statistical Evaluation* task collects all data products generated from CM, RC and SMC to produce statistical results using the t-distribution. The statistical results can be visualized by the *Visualization* task, using the MRI as the spatial context with the affected tissues labeled and the statistical variations colored with different colormaps.

The service-oriented architecture of VIEW enables a fast and convenient development of the FiberFlow system, by customizing subsystem GUIs, while reusing the underlying VIEW Kernel. Some customizations are performed on the FiberFlow Workbench without changing any source codes of the VIEW Workbench (see Figure 6.5 (right)). For example, the VIEW Workbench provides the options for users to choose which subsystem GUI to be displayed on the control panel. Users can also customize icons, menus, font size and color in the workflow design panel to make the look-and-feel consistent with other non-workflow based subsystems embedded in FiberFlow. We plan to collect more use case scenarios to improve the flexibility and configurability of customizing the VIEW Workbench for different scientific domains. By reusing the VIEW Kernel, the Task Manager and the Data Product Manager manage a library of software tools and data products for this domain. The Workflow Engine manages our entire set of FiberFlow workflows composed by existing software tools and data products. The Workflow Monitor is used to monitor workflow execution progress, and the Provenance Manager is employed for provenance management, on which provenance mining and provenance visualization are being conducted for our domain-specific analysis.

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

This dissertation presented an integrated solution to composing, scheduling, executing and developing scientific workflows and scientific workflow management systems. To provide a foundation for workflow composition, scheduling, execution and management, we proposed the *first* reference architecture for scientific workflow management systems, which is composed of four logical layers, seven major functional subsystems, and six interfaces. The reference architecture not only provides a high-level organization of subsystems and their interactions in a workflow system, but also provides a basis for comparison between different systems and a guidance for the architectural design of developing an SWFMS in a specific scientific domain.

To integrate heterogeneous services and applications into workflows, we proposed a task template model which not only provides an appropriate abstraction of heterogeneous services and applications, but also encapsulates the composition and mapping of shims and functional task components within a task interface. We designed an XML-based task specification language, called TSL, to realize the proposed task template model. TSL not only enables the abstraction of heterogeneous services and applications into uniform workflow tasks, but also provides a solution to address both TYPE-I and TYPE-II shimming problems in composing scientific workflows. To our best knowledge, this is the *first* shimming technique that makes shims invisible at the workflow level, resulting in scientific workflows that are more elegant and readable.

To schedule scientific workflows in emerging services computing environments, we proposed two workflow scheduling algorithms, the SHEFT algorithm and SCPOR algorithm, to prioritize tasks in a workflow, map tasks onto suitable resources and order the execution of tasks on the assigned resources, so that the workflow makespan can be minimized. Our extensive experiments showed that our proposed algorithms not only outperform the HEFT and CPOP algorithms for data-intensive and compute intensive workflows, but also allow the assigned resources elastically change on demand of the scalability of workflows.

To execute scientific workflows on distributed and heterogeneous computing environments, we proposed a task run model to model the run-time behaviors of tasks. Based on the task run model, we designed the task run description language, TRDL, for the description of task runs, to support the execution of task instances constructed from heterogeneous services and applications. We also developed an SOA based task management subsystem, the Task Manager, to manage all task templates, task instances and task runs for the invocation and execution of various heterogeneous task components.

Finally, our developed VIEW scientific workflow management system and a VIEW based workflow application system, the FiberFlow system, validate our architectures, models, languages and algorithms.

7.2 Future Work

We foresee many improvements, extensions, and applications of our current research work. Possible future research work which I am particularly interested in is listed as follows.

Workflow Computing on the Cloud. Addressing complex scientific problems requires analyzing massive datasets using cluster-based and high-performance distributed computing. Managing distributed scientific workflows that can run concurrently on multiple clusters or computers remains a significant challenge. The ability of cloud computing to scale on demand as usage changes, through dynamic provisioning and configuration of integrated virtual machines, offers appealing opportunities for SWFMSs to address this issue. We will endeavor to develop on-cloud workflow services for scientists to manage distributed scientific workflows and perform massively parallel data processing in Cloud environments. In order to fulfill this plan, we have identified several key research problems for my future research: (1) Cost models are needed to optimize workflow scheduling in a Cloud environment; (2) Intelligent and dynamic scheduling algorithms are needed to determine the optimal placement of data and computation in a Cloud computing environment; (3) Workflow monitoring framework is required to support the reliability of distributed workflow execution and management.

e-Science Workflow Applications and Systems. Through all these years of collaboration with scientists in multiple domains, we believe that new levels of understanding and knowledge about scientific processes could underpin new challenges of computing technology. Therefore, we have strong interests in adapting computational techniques to problems in a wide spectrum of scientific domains, and in developing complex workflow application systems that are resilient, fault tolerant, adaptive, and learning.

APPENDIX A: TASK SPECIFICATION LANGUAGE (TSL)

```
<?xml␣version="1.0"␣encoding="UTF-8"?>
<xs:schema␣xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:dt="http://http://database.cs.wayne.edu/DPM/XMLSchema"
elementFormDefault="qualified"␣attributeFormDefault="unqualified"
version="1.0">

<xs:element␣name="taskTemplate"␣type="taskTemplate_XMLElementType"/>
<xs:complexType␣name="taskTemplate_XMLElementType">
<xs:sequence>
<xs:element␣name="taskInterface"␣type="taskInterface_XMLElementType"/>
<xs:element␣name="taskComponents"␣type="taskComponents_XMLElementType"/>
<xs:element␣name="mappings"␣type="mappings_XMLElementType"/>
<xs:element␣name="taskInstances"␣type="taskInstances_XMLElementType"/>
</xs:sequence>
<xs:attribute␣name="version"␣type="xs:string"␣use="required"/>
</xs:complexType>

<xs:complexType␣name="taskInterface_XMLElementType">
<xs:sequence>
<xs:element␣name="taskName"␣type="xs:string"/>
<xs:element␣name="taskDescription"␣type="xs:string"/>
<xs:element␣name="inputPorts"␣type="inputPorts_XMLElementType"
minOccurs="0"␣maxOccurs="unbounded"/>
<xs:element␣name="outputPorts"␣type="outputPort_XMLElementType"
minOccurs="0"␣maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute␣name="id"␣type="xs:NMTOKEN"␣use="required"/>
</xs:complexType>

<xs:complexType␣name="inputPorts_XMLElementType">
<xs:sequence>
<xs:element␣name="port"␣type="port_XMLElementType"
maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute␣name="number"␣type="xs:NMTOKEN"␣use="required"/>
</xs:complexType>

<xs:complexType␣name="outputPort_XMLElementType">
<xs:sequence>
<xs:element␣name="port"␣type="port_XMLElementType"
maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute␣name="number"␣type="xs:int"␣use="required"/>
```

```
</xs:complexType>

<xs:complexType␣name="port_XMLElementType">
<xs:sequence>
<xs:element␣name="portType"␣type="xs:string"/>
<xs:element␣name="portDescription"␣type="xs:string"␣minOccurs="0"/>
<xs:element␣name="defaultPortValue"␣minOccurs="0"/>
</xs:sequence>
<xs:attribute␣name="id"␣type="xs:NMTOKEN"␣use="required"/>
<xs:attribute␣name="default"␣type="xs:NMTOKEN"␣use="optional"/>
</xs:complexType>

<xs:complexType␣name="taskComponents_XMLElementType">
<xs:sequence>
<xs:element␣ref="taskComponent"␣maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:element␣name="taskComponent">
<xs:complexType>
<xs:sequence>
<xs:element␣name="taskType">
<xs:simpleType>
<xs:restriction␣base="xs:string">
<xs:enumeration␣value="WebService"/>
<xs:enumeration␣value="WindowsApplication"/>
<xs:enumeration␣value="UnixApplication"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:choice>
<xs:group␣ref="WebServiceGroup"/>
<xs:group␣ref="WindowsApplicationGroup"/>
<xs:group␣ref="UnixApplicationGroup"/>
</xs:choice>
<xs:element␣name="taskInvocation"␣type="taskInvocation_XMLElementType"/>
<xs:element␣name="inputs"␣type="inputs_XMLElementType"/>
<xs:element␣name="outputs"␣type="outputs_XMLElementType"/>
</xs:sequence>
<xs:attribute␣name="id"␣type="xs:NMTOKEN"␣use="required"/>
<xs:attribute␣name="default"␣type="xs:NMTOKEN"␣use="required"/>
<xs:attribute␣ref="role"␣use="required"/>
</xs:complexType>
</xs:element>

<xs:attribute␣name="role">
```

```
<xs:simpleType>
<xs:restriction␣base="xs:string">
<xs:enumeration␣value="functional"/>
<xs:enumeration␣value="shim"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>

<xs:group␣name="WebServiceGroup">
<xs:sequence>
<xs:element␣name="wsdlURI"␣type="xs:string"/>
<xs:element␣name="serviceName"␣type="xs:string"/>
<xs:element␣name="operationName"␣type="xs:string"/>
</xs:sequence>
</xs:group>

<xs:group␣name="WindowsApplicationGroup">
<xs:sequence>
<xs:element␣name="executable"␣type="xs:string"/>
<xs:element␣name="appName"␣type="xs:string"/>
</xs:sequence>
</xs:group>

<xs:group␣name="UnixApplicationGroup">
<xs:sequence>
<xs:element␣name="directory"␣type="xs:string"/>
<xs:element␣name="appName"␣type="xs:string"/>
</xs:sequence>
</xs:group>

<xs:complexType␣name="taskInvocation_XMLElementType">
<xs:sequence>
<xs:element␣name="operatingSystem">
<xs:simpleType>
<xs:restriction␣base="xs:string">
<xs:enumeration␣value="Windows"/>
<xs:enumeration␣value="UNIX"/>
<xs:enumeration␣value="LINUX"/>
<xs:enumeration␣value="MAC"/>
<xs:enumeration␣value="Unknown"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element␣name="invocationMode">
<xs:simpleType>
<xs:restriction␣base="xs:string">
```

```
<xs:enumeration␣value="Local"/>
<xs:enumeration␣value="Remote"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element␣name="interactionMode">
<xs:simpleType>
<xs:restriction␣base="xs:string">
<xs:enumeration␣value="Yes"/>
<xs:enumeration␣value="No"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element␣ref="invocationAuthentication"
minOccurs="0"␣maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>
<xs:element␣name="invocationAuthentication">
<xs:complexType>
<xs:sequence>
<xs:element␣name="hostName"␣type="xs:string"/>
<xs:element␣name="userName"␣type="xs:string"/>
<xs:element␣name="password"␣type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:complexType␣name="inputs_XMLElementType">
<xs:sequence>
<xs:element␣ref="input"␣maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType␣name="outputs_XMLElementType">
<xs:sequence>
<xs:element␣ref="output"␣maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:element␣name="input">
<xs:complexType>
<xs:attribute␣name="id"␣type="xs:NMTOKEN"␣use="required"/>
<xs:attribute␣name="mode"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="name"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="type"␣type="xs:string"␣use="required"/>
</xs:complexType>
```

```
</xs:element>

<xs:element␣name="output">
<xs:complexType>
<xs:attribute␣name="id"␣type="xs:NMTOKEN"␣use="required"/>
<xs:attribute␣name="mode"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="name"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="type"␣type="xs:string"␣use="required"/>
</xs:complexType>
</xs:element>

<xs:complexType␣name="mappings_XMLElementType">
<xs:sequence>
<xs:element␣name="mapping"␣type="mapping_XMLElementType"
maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType␣name="mapping_XMLElementType">
<xs:sequence>
<xs:element␣name="inputmapping"␣type="inputmapping_XMLElementType"
minOccurs="0"␣maxOccurs="unbounded"/>
<xs:element␣ref="outputmapping"␣minOccurs="0"␣maxOccurs="unbounded"/>
<xs:element␣ref="assign"␣minOccurs="0"␣maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute␣name="id"␣type="xs:NMTOKEN"␣use="required"/>
</xs:complexType>

<xs:complexType␣name="inputmapping_XMLElementType">
<xs:sequence>
<xs:element␣name="shims"␣type="shims_XMLElementType"
minOccurs="0"␣maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute␣name="from"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="to"␣type="xs:string"␣use="required"/>
<xs:attribute␣ref="shimming"␣use="required"/>
</xs:complexType>

<xs:attribute␣name="shimming">
<xs:simpleType>
<xs:restriction␣base="xs:string">
<xs:enumeration␣value="Yes"/>
<xs:enumeration␣value="No"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>
```

```
<xs:complexType␣name="shims_XMLElementType">
<xs:sequence>
<xs:element␣ref="shimming"/>
</xs:sequence>
<xs:attribute␣name="id"␣type="xs:string"␣use="required"/>
</xs:complexType>

<xs:element␣name="shimming">
<xs:complexType>
<xs:attribute␣name="from"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="to"␣type="xs:string"␣use="required"/>
</xs:complexType>
</xs:element>

<xs:element␣name="outputmapping">
<xs:complexType>
<xs:attribute␣name="from"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="to"␣type="xs:string"␣use="required"/>
</xs:complexType>
</xs:element>

<xs:element␣name="assign">
<xs:complexType>
<xs:attribute␣name="from"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="to"␣type="xs:string"␣use="required"/>
</xs:complexType>
</xs:element>

<xs:complexType␣name="taskInstances_XMLElementType">
<xs:sequence>
<xs:element␣name="taskInstance"␣type="taskInstance_XMLElementType"
minOccurs="0"␣maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType␣name="taskInstance_XMLElementType">
<xs:sequence>
<xs:element␣name="taskComponent"␣type="taskComponent_XMLElementType"/>
</xs:sequence>
<xs:attribute␣name="id"␣type="xs:NMTOKEN"␣use="required"/>
</xs:complexType>
<xs:complexType␣name="taskComponent_XMLElementType">
<xs:attribute␣name="id"␣type="xs:int"␣use="required"/>
</xs:complexType>
</xs:schema>
```

APPENDIX B: TASK RUN DESCRIPTION LANGUAGE (TRDL)

```
<?xml␣version="1.0"␣encoding="UTF-8"?>
<xs:schema␣xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"␣attributeFormDefault="unqualified"
version="1.0">

<xs:element␣name="taskRun"␣type="taskRun_XMLElementType"/>
<xs:complexType␣name="taskRun_XMLElementType">
<xs:sequence>
<xs:element␣name="taskRunInterface"
type="taskRunInterface_XMLElementType"/>
<xs:element␣ref="taskComponent"/>
<xs:element␣name="bindings"␣type="bindings_XMLElementType"/>
</xs:sequence>
<xs:attribute␣name="version"␣type="xs:string"␣use="required"/>
</xs:complexType>

<xs:complexType␣name="taskRunInterface_XMLElementType">
<xs:sequence>
<xs:element␣name="taskTemplate_ID"␣type="xs:int"/>
<xs:element␣name="taskComponent_ID"␣type="xs:int"/>
<xs:element␣name="taskInstance_ID"␣type="xs:int"/>
<xs:element␣name="workflowRun_ID"␣type="xs:int"/>
<xs:element␣name="taskRun_State"␣type="xs:string"/>
<xs:element␣name="inputPorts"␣type="inputPorts_XMLElementType"
minOccurs="0"␣maxOccurs="unbounded"/>
<xs:element␣name="outputPorts"␣type="outputPort_XMLElementType"
minOccurs="0"␣maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute␣name="id"␣type="xs:int"␣use="required"/>
</xs:complexType>

<xs:complexType␣name="inputPorts_XMLElementType">
<xs:sequence>
<xs:element␣name="port"␣type="port_XMLElementType"
maxOccurs="unbounded"/>
</xs:sequence>
<xs:attribute␣name="number"␣type="xs:NMTOKEN"␣use="required"/>
</xs:complexType>

<xs:complexType␣name="outputPort_XMLElementType">
<xs:sequence>
<xs:element␣name="port"␣type="port_XMLElementType"
maxOccurs="unbounded"/>
```

```
</xs:sequence>
<xs:attribute␣name="number"␣type="xs:int"␣use="required"/>
</xs:complexType>

<xs:complexType␣name="port_XMLElementType">
<xs:sequence>
<xs:element␣name="portType"␣type="xs:string"/>
<xs:element␣name="portDescription"␣type="xs:string"
minOccurs="0"/>
<xs:element␣name="defaultPortValue"␣minOccurs="0"/>
</xs:sequence>
<xs:attribute␣name="id"␣type="xs:NMTOKEN"␣use="required"/>
<xs:attribute␣name="default"␣type="xs:NMTOKEN"␣use="optional"/>
</xs:complexType>

<xs:element␣name="taskComponent">
<xs:complexType>
<xs:sequence>
<xs:element␣name="taskType">
<xs:simpleType>
<xs:restriction␣base="xs:string">
<xs:enumeration␣value="WebService"/>
<xs:enumeration␣value="WindowsApplication"/>
<xs:enumeration␣value="UnixApplication"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:choice>
<xs:group␣ref="WebServiceGroup"/>
<xs:group␣ref="WindowsApplicationGroup"/>
<xs:group␣ref="UnixApplicationGroup"/>
</xs:choice>
<xs:element␣name="taskInvocation"
type="taskInvocation_XMLElementType"/>
<xs:element␣name="inputs"␣type="inputs_XMLElementType"/>
<xs:element␣name="outputs"␣type="outputs_XMLElementType"/>
</xs:sequence>
<xs:attribute␣name="id"␣type="xs:NMTOKEN"␣use="required"/>
<xs:attribute␣name="default"␣type="xs:NMTOKEN"␣use="required"/>
<xs:attribute␣ref="role"␣use="required"/>
</xs:complexType>
</xs:element>

<xs:attribute␣name="role">
<xs:simpleType>
<xs:restriction␣base="xs:string">
```

```
<xs:enumeration␣value="Yes"/>
<xs:enumeration␣value="No"/>
</xs:restriction>
</xs:simpleType>
</xs:attribute>

<xs:group␣name="WebServiceGroup">
<xs:sequence>
<xs:element␣name="wsdlURI"␣type="xs:string"/>
<xs:element␣name="serviceName"␣type="xs:string"/>
<xs:element␣name="operationName"␣type="xs:string"/>
</xs:sequence>
</xs:group>

<xs:group␣name="WindowsApplicationGroup">
<xs:sequence>
<xs:element␣name="executable"␣type="xs:string"/>
<xs:element␣name="appName"␣type="xs:string"/>
</xs:sequence>
</xs:group>

<xs:group␣name="UnixApplicationGroup">
<xs:sequence>
<xs:element␣name="directory"␣type="xs:string"/>
<xs:element␣name="appName"␣type="xs:string"/>
</xs:sequence>
</xs:group>

<xs:complexType␣name="taskInvocation_XMLElementType">
<xs:sequence>
<xs:element␣name="operatingSystem">
<xs:simpleType>
<xs:restriction␣base="xs:string">
<xs:enumeration␣value="Windows"/>
<xs:enumeration␣value="UNIX"/>
<xs:enumeration␣value="LINUX"/>
<xs:enumeration␣value="MAC"/>
<xs:enumeration␣value="Unknown"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element␣name="invocationMode">
<xs:simpleType>
<xs:restriction␣base="xs:string">
<xs:enumeration␣value="Yes"/>
<xs:enumeration␣value="No"/>
```

```
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element␣name="interactionMode">
<xs:simpleType>
<xs:restriction␣base="xs:string">
<xs:enumeration␣value="Local"/>
<xs:enumeration␣value="Remote"/>
</xs:restriction>
</xs:simpleType>
</xs:element>
<xs:element␣ref="invocationAuthentication"␣minOccurs="0"
maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:element␣name="invocationAuthentication">
<xs:complexType>
<xs:sequence>
<xs:element␣name="hostName"␣type="xs:string"/>
<xs:element␣name="userName"␣type="xs:string"/>
<xs:element␣name="password"␣type="xs:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>

<xs:complexType␣name="inputs_XMLElementType">
<xs:sequence>
<xs:element␣ref="input"␣maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType␣name="outputs_XMLElementType">
<xs:sequence>
<xs:element␣ref="output"␣maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:element␣name="input">
<xs:complexType>
<xs:attribute␣name="id"␣type="xs:NMTOKEN"␣use="required"/>
<xs:attribute␣name="mode"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="name"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="type"␣type="xs:string"␣use="required"/>
</xs:complexType>
</xs:element>
```

```
<xs:element␣name="output">
<xs:complexType>
<xs:attribute␣name="id"␣type="xs:NMTOKEN"␣use="required"/>
<xs:attribute␣name="mode"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="name"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="type"␣type="xs:string"␣use="required"/>
</xs:complexType>
</xs:element>

<xs:complexType␣name="bindings_XMLElementType">
<xs:sequence>
<xs:element␣name="data_inputport_binding"
type="binding_XMLElementType"␣maxOccurs="unbounded"/>
<xs:element␣name="data_input_binding"
type="binding_XMLElementType"␣maxOccurs="unbounded"/>
<xs:element␣name="output_data_binding"
type="binding_XMLElementType"␣maxOccurs="unbounded"/>
<xs:element␣name="data_outputport_binding"
type="binding_XMLElementType"␣maxOccurs="unbounded"/>
</xs:sequence>
</xs:complexType>

<xs:complexType␣name="binding_XMLElementType">
<xs:attribute␣name="from"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="to"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="timestamp"␣type="xs:string"␣use="required"/>
<xs:attribute␣name="dataValue"␣type="xs:string"␣use="required"/>
</xs:complexType>
</xs:schema>
```

REFERENCES

[1] D. Hollingsworth, “The workflow reference model,” *The Workflow Management Coalition*, 1994.

[2] I.J. Taylor, E. Deelman, D.B. Gannon, and M. Shields, *Workflows for e-science*, Springer-Verlag London Limited, 2007.

[3] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E.A. Lee, J. Tao, and Y. Zhao, “Scientific workflow management and the Kepler system,” *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2006.

[4] S.P. Callahan, J. Freire, E. Santos, C.E. Scheidegger, C.T. Silva, and H.T. Vo, “VisTrails: visualization meets data management,” in *ACM SIGMOD international conference on Management of data*, 2006, pp. 745–747.

[5] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G.V. Laszewski, I. Raicu, T. Stef-Praun, and M. Wilde, “Swift: Fast, reliable, loosely coupled parallel computation,” in *IEEE International Workshop on Scientific Workflows*, 2007, pp. 199–206.

[6] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G.B. Berriman, J. Good, A. Laity, J.C. Jacob, and D.S. Katz, “Pegasus: A framework for mapping complex scientific workflows onto distributed systems,” *Scientific Programming Journal*, vol. 13, no. 3, pp. 219–237, 2005.

[7] T.M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R.M. Greenwood, T. Carver, K. Glover, M.R. Pocock, A. Wipat, and P. Li, “Taverna: A tool for the composition and enactment of bioinformatics workflows,” *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[8] S. Majithia, M.S. Shields, I.J. Taylor, and I. Wang, “Triana: A graphical web service composition and execution toolkit,” in *IEEE International Conference on Web Services*, 2004, pp. 514–524.

[9] P. Grefen and R.R. de Vries, “A reference architecture for workflow management systems,” *Data and Knowledge Engineering*, vol. 27, no. 1, pp. 31–57, 1998.

[10] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede, "Design and implementation of the YAWL system," in *International Conference on Advanced Information Systems Engineering*, 2004, pp. 142–159.

[11] L. Liu, C. Pu, and D.D.A. Ruiz, "A systematic approach to flexible specification, composition, and restructuring of workflow activities," *Journal of Database Management*, vol. 15, no. 1, pp. 1–40, 2004.

[12] J. A. Miller, D. Palaniswami, A. P. Sheth, K. Kochut, and H. Singh, "Webwork: METEOR_2's web-based workflow management system," *Journal of Intelligent Information Systems*, vol. 10, no. 2, pp. 185–215, 1998.

[13] G. Alonso, R. Günthör, M. Kamath, D. Agrawal, A.E. Abbadi, and C. Mohan, "Exotica/FMDC: A workflow management system for mobile and disconnected clients," *Distributed and Parallel Databases*, vol. 4, no. 3, pp. 229–247, 1996.

[14] F. Leymann and D. Roller, "Business process management with FlowMark," in *COMPCON*, 1994, pp. 230–234.

[15] J.L. Ambite and D. Kapoor, "Automatically composing data workflows with relational descriptions and shim services," in *6th International and 2nd Asian Semantic Web Conference*, 2007, pp. 15–28.

[16] B. Ludäscher, S. Bowers, T.M. McPhillips, and N. Podhorszki, "Scientific workflows: More e-science mileage from cyberinfrastructure," in *International Conference on e-Science and Grid Computing*, 2006, pp. 145–152.

[17] D. Hull, R. Stevens, P. Lord, C. Wroe, and C. Goble, "Treating shimantic web syndrome with ontologies," in *AKT Workshop on Semantic Web Services*, 2004.

[18] J.D. Ullman, "NP-complete scheduling programs," *Computer and Systems Sciences*, vol. 10, pp. 384–393, 1975.

[19] M.R.Garey and D.S.Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA, 1990.

[20] F. Dong and S.G. Akl, "Scheduling algorithms for Grid computing: State of the art and open problems," Tech. Rep. No. 2006-504, Queens University, 2006.

[21] D. Tombros, *An Event- and Repository-Based Component Framework for Workflow System Architecture*, Phd thesis, Department of Information Technology, University of Zurich, 1999.

[22] M. Hsu and C. Kleissner, "Objectflow: Towards a distributed process management infrastructure," *Distributed and Parallel Databases*, vol. 4, no. 2, pp. 169, 1996.

[23] G. Kappel, B. Proll, S. Rausch-Schott, and W. Retschitzegger, "TriGSflow - active object-oriented workflow management," in *28th Hawaii international conference on system sciences*, January, 1995, pp. 12–26.

[24] H. Schuster, S. Jablonski, T. Kirsche, and C. Bussler, "A client/server architecture for distributed workflow management systems," in *Parallel and Distributed Information Systems*, 1994, pp. 253–256.

[25] C. Fernstrom, "Process WEAVER: Adding process support to UNIX," in *International Conference on Software Process*, 1993, pp. 12–26.

[26] P. Heinl and H. Schuster, "Towards a highly scalable architecture of workflow management systems," in *7th International Conference on Database and Expert System Applications*, 1996.

[27] H. Schuster, S. Jablonski, P. Heinl, and C. Bussler, "A general framework for the execution of heterogenous programs in workflow management systems," in *First IFCIS International Conference on Cooperative Information Systems*, 1996, pp. 104 – 113.

[28] G. Alonso, C. Mohan, R. Gunthor, D. Agrawal, A.E. Abbadi, and M. Kamath, "Exotica/fmqm:a persistent message-based architecture for distributed workflow management," in *IFIP Working Conference on Information Systems for Decentralized Organizations,Trondheim, Norway*, 1995.

[29] J. Miller, A. Sheth, K. Kochut, and X.Wang, “CORBA-based runtime architectures for workflow management systems,” *Journal of database management, special issue on multidatabases*, vol. 7, no. 1, pp. 16–27, 1996.

[30] F. Leyman and W. Altenhuber, “Managing business processes as an information resource,” *IBM Systems Journal*, vol. 33, no. 2, pp. 36–47, 1994.

[31] G. Alonso, B. Reinwald, and C. Mohan, “Wide - a distributed architecture for workflow management,” in *Seventh International Workshop on Research Issues in Data Engineering*, 1997, pp. 76–79.

[32] M. Cagan, “The HP softBench environment: an architecture for a new generation of software tools,” *Hewlett-Packard Journal*, vol. 41, no. 3, pp. 36–47, 1990.

[33] K.J. Kochut, A.P. Sheth, J.A. Miller, S. Das, and Z. Luo, “ORBWork: A Dynamic Workflow Enactment Service for METEOR$_2$,” *Technical report, University of Georgia*, 1997.

[34] Z. Mahmood, “Service oriented architecture: a new paradigm for enterprise application integration,” in *11th WSEAS International Conference on Computers*, 2007, pp. 491–496.

[35] Y. Zhao, Y. Feng, and H. Liu, “Research on service-oriented workflow management system architecture,” in *Ninth International Conference on Hybrid Intelligent Systems*, 2009, pp. 369–372.

[36] W.M.P. van der Aalst, L. Aldred, M. Dumas, and A.H.M. ter Hofstede, “Design and implementation of the YAWL system,” in *16th International Conference on Advanced Information Systems Engineering*, 2004, pp. 142–159.

[37] W.M.P. van der Aalst and A.H.M. ter Hofstede, “YAWL: yet another workflow language,” *Information Systems*, vol. 30, no. 4, pp. 245–275, 2005.

[38] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao, and Y. Zhao, “Scientific workflow management and the Kepler system,” *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1039–1065, 2005.

[39] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *16th International Conference on Scientific and Statistical Database Management*, June, 2004, pp. 423– 424.

[40] S. Bowers and B. Ludäscher, "Actor-oriented design of scientific workflows," in *24st International Conference on Conceptual Modeling*. 2005, Springer.

[41] I. Altintas, A. Birnbaum, K. Baldridge, W. Sudholt, M. Miller, C. Amoreira, Y. Potier, and B. Ludäscher, "A framework for the design and reuse of Grid workflows," in *International Workshop on Scientific Applications on Grid Computing*, 2005, pp. 120–133.

[42] Ptolemy II, ," in *http://ptolemy.eecs.berkeley.edu/ptolemyII/*.

[43] X. Liu, J. Liu, J.Eker, and E.A. Lee, *Heterogeneous Modeling and Design of Control System*, Software-Enabled Control: Information Technology for Dynamical Systems, April 2003.

[44] T. Oinn, M. Greenwood, M. J. Addis, M. N. Alpdemir, J.Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. J. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "Taverna: Lessons in creating a workflow environment for the life sciences," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 10, pp. 1067–1100, 2005.

[45] T. Oinn, M.J. Addis, J. Ferris, D.J. Marvin, M. Senger, T. Carver, M. Greenwood, K. Glover, M.R. Pocock, A. Wipat, and P. Li, "Taverna: a tool for the composition and enactment of bioinformatics workflows," *Bioinformatics*, vol. 20, no. 17, pp. 3045–3054, 2004.

[46] I. Taylor, M. Shields, I. Wang, and R. Philp, "Distributed P2P computing within Triana: A galaxy visualization test case," *17th International Parallel and Distributed Processing Symposium*, pp. 16–27, 2003.

[47] I. Taylor, M. Shields, I. Wang, and A. Harrison, "The Triana Workflow Environment: Architecture and Applications," in *Workflows for e-Science*, pp. 320–339. Springer, 2007.

[48] S. Majithia, M.S. Shields, I.J. Taylor, and I. Wang, "Triana: A Graphical Web Service Composition and Execution Toolkit," in *IEEE International Conference on Web Services*, 2004, pp. 514–524.

[49] E. Gallopoulos, E.N. Houstis, and J.R. Rice, "Computer as thinker/doer: Problem solving environments for computational science," *IEEE Computer Science and Engineering*, vol. 1, no. 2, pp. 11–23, 1994.

[50] G. Allen, T. Goodale, T. Radke, M. Russell, E. Seidel, K. Davis, K.N. Dolkas, N.D. Doulamis, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, J. Shalf, and I. Taylor, "Enabling applications on the Grid: A GridLab overview," *International Journal of high performance computing applications: special issue on Grid computing: infrastructure and applications*, vol. 17, no. 4, pp. 449–466, 2003.

[51] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil1, M. Su, K. Vahi, and M. Livny, "Pegasus: Mapping scientificworkflows onto the Grid," in *2nd European across Grids conference*, 2004.

[52] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, K. Blackburn, A. Lazzarini, A. Arbree, R. Cavanaugh, and S. Koranda, "Mapping abstract complex workflows onto *grid* environments," *Journal of Grid computing*, vol. 1, no. 1, pp. 25–39, 2003.

[53] E. Deelman, J. Blythe, Y. Gil, and C. Kesselman, "Workflow management in GriPhyN," *Grid resource management: state of the art and future trends*, pp. 99–116, 2004.

[54] I. Foster, C. Kesselman, J.M. Nick, and S. Tuecke, "Grid services for distributed system integration," *Computer*, vol. 35, no. 6, pp. 37–46, 2002.

[55] T. Tannenbaum, D. Wright, K. Miller, and M. Livny, "Condor: a distributed job scheduler," *Beowulf cluster computing with Linux*, pp. 307–350, 2002.

[56] N. Krishnakumar and A. Sheth, "Managing heterogeneous multi-system tasks to support enterprise-wide operations," *Distributed and parallel databases*, vol. 3, no. 2, pp. 155–186, 1995.

[57] H. Schuster, S. Jablonski, P. Heinl, and C. Bussler, "A general framework for the execution of heterogeneous programs in workflow management systems," in *International Conference on Cooperative Information Systems*, 1996, p. 104.

[58] P. Karagoz, S. Arpinar, P. Koksal, N. Tatbul, E. Gokkoca, and A. Dogac, "Task handling in workflow management systems," in *International Workshop on Issues and Applications of Database Technology*, 1998.

[59] J.-Y. Jung, H. Kim, and S.-H. Kang, "Standards-based approaches to b2b workflow integration," *Computers and Industrial Engineering*, vol. 51, no. 2, pp. 321–334, 2006.

[60] C. Walker and D. W. Walker, "Integration and data sharing between ws-based workflows," in *IEEE International Conference on Web Services*, 2008, pp. 667–674.

[61] H. Weigand and W.-J. van den Heuvel, "Cross-organizational workflow integration using contracts," *Decision Support Systems*, vol. 33, no. 3, pp. 247–265, 2002.

[62] W.M.P. van der Aalst, A.H.M. ter Hofstede, B. Kiepuszewski, and A.P. Barros, "Workflow patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[63] C. Lin, S. Lu, Z. Lai, A. Chebotko, X. Fei, J. Hua, and F. Fotouhi, "Service-oriented architecture for VIEW: A visual scientific workflow management system," in *IEEE SCC*, 2008, pp. 335–342.

[64] C. Lin, S. Lu, X. Fei, A. Chebotko, Z. Lai, D. Pai, F. Fotouhi, and J. Hua, "A reference architecture for scientific workflow management systems and the VIEW SOA solution," *IEEE Transactions on Services Computing*, vol. 2, no. 1, pp. 79–92, 2009.

[65] A. Tsalgatidou, G. Athanasopoulos, M. Pantazoglou, C. Pautasso, T. Heinis, R. Grønmo, H. Hoff, A. Berre, M. Glittum, and S. Topouzidou, "Developing scientific workflows from heterogeneous services," *SIGMOD Record*, vol. 35, no. 2, pp. 22–28, 2006.

[66] M. Szomszor, T.R. Payne, and L.Moreau, "Automated syntactic medation for web service integration," in *ICWS*, 2006, pp. 127–136.

[67] D. Hull, R. Stevens, and P. Lord, “Describing web services for user-oriented retrieval,” in *W3C Workshop on Frameworks for Semantics in Web Services*, 2005, pp. 9–10.

[68] R. Armstrong, D. Hensgen, and T. Kidd, “The relative performance of various mapping algorithms is independent of sizable variances in run-time predictions,” in *7th IEEE Heterogeneous Computing Workshop*, 1998, pp. 79–87.

[69] R.F. Freund, M. Gherrity, S. Ambrosius, M. Campbell, M. Halderman, D. Hensgen, E. Keith, T. Kidd, M. Kussow, J.D. Lima, F. Mirabile, L. Moore, B. Rust, and H.J. Siegel, “Scheduling resources in multi-user, heterogeneous, computing environments with smartnet,” in *7th IEEE Heterogeneous Computing Workshop*, 1998, pp. 184–199.

[70] M. Wieczorek, R. Prodan, and T. Fahringer, “Scheduling of scientific workflows in the askalon Grid environment,” *SIGMOD Rec.*, vol. 34, no. 3, pp. 56–62, 2005.

[71] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. Freund, “Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems,” in *8th Heterogeneous Computing Workshop*, 1999, p. 30.

[72] T.D. Braun, H.J. Siegel, N. Beck, L.L. Boloni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M. Theys, B. Yao, D. Hensgen, and R.F. Freund, “A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems,” *Journal of Parallel and Distributed Computing*, vol. 61, no. 6, pp. 810–837, 2001.

[73] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*, MIT Press, Cambridge, MA, USA, 1989.

[74] T. Yang and A. Gerasoulis, “A fast static scheduling algorithm for dags on an unbounded number of processors,” in *ACM/IEEE conference on Supercomputing*, 1991, pp. 633–642.

[75] A. Gerasoulis and T. Yang, “A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors,” *Journal of Parallel and Distributed Computing*, vol. 16, no. 4, pp. 276–291, 1992.

[76] T. Yang and A. Gerasoulis, "Dsc: Scheduling parallel tasks on an unbounded number of processors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, pp. 951–967, 1994.

[77] M.A. Palis, J.-C. Liou, and D.S. Wei, "Task clustering and scheduling for distributed memory parallel architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, no. 1, pp. 46–55, 1996.

[78] J.C. Liou and M.A. Palis, "An efficient task clustering heuristic for scheduling dags on multiprocessors," in *Workshop on Resource Management, Symposium of Parallel and Distributed Processing*, 1996, pp. 152–156.

[79] S.J. Kim and J.C. Browne, "A general approach to mapping of parallel computation upon multiprocessor architectures," in *International Conference on Parallel Processing*, 1988, pp. 1–8.

[80] J. Liou and M.A. Palis, "A comparison of general approaches to multiprocessor scheduling," in *11th International Symposium on Parallel Processing*, 1997, pp. 152–156.

[81] R. Bajaj and D.P. Agrawal, "Improving scheduling of tasks in a heterogeneous environment," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, pp. 107–118, 2004.

[82] I. Ahmad and Y. Kwok, "A new approach to scheduling parallel processsing," in *International Conference on Parallel Processing*, 2007, pp. 539–550.

[83] B. Kruatrachue and T.G. Lewis, "Grain size determination for parallel processing," *IEEE Software*, vol. 5, no. 1, pp. 23–32, 1988.

[84] Y. Chung and S. Ranka, "Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory multiprocessors," in *Supercomputing*, 1992, pp. 512–521.

[85] K. Ranganathan and I. Foster, "Decoupling computation and data scheduling in distributed data-intensive applications," in *11th IEEE International Symposium on High Performance Distributed Computing*, 2002, p. 352.

[86] K. Ranganathan and I. Foster, "Identifying dynamic replication strategies for a high-performance data Grid," in *2nd International Workshop on Grid Computing*, 2001, pp. 75–86.

[87] H. El-Rewini and T.G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *journal of parallel and distributed computing*, vol. 9, no. 2, pp. 138–153, 1990.

[88] G.C. Sih and E.A. Lee, "A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 2, pp. 175–187, 1993.

[89] Y.K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Computer Surveys*, vol. 31, no. 4, pp. 406–471, 1999.

[90] M.Y Wu and D.D. Gajski, "A programming aid for hypercube architectures," *The Journal of Supercomputing*, vol. 2, no. 3, pp. 349–372, 1988.

[91] B. Kruatrachue and T. Lewis, "Grain size determination for parallel processing," *IEEE Software*, vol. 5, no. 1, pp. 23–32, 1988.

[92] J.J. Hwang, Y.-C. Chow, F.D. Anger, and C.-Y. Lee, "Scheduling precedence graphs in systems with interprocessor communication times," *SIAM Journal on Computing*, vol. 18, no. 2, pp. 244–257, 1989.

[93] Y. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 7, pp. 506–521, 1996.

[94] H. Topcuoglu, S. Hariri, and M. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, pp. 260–274, 2002.

[95] H. El-Rewini and T.G. Lewis, "Scheduling parallel program tasks onto arbitrary target machines," *Journal of Parallel and Distributed Computing*, vol. 9, no. 2, pp. 138–153, 1990.

[96] M.A. Iverson, F. zgner, O. Gregory, and G.J. Follen, "Parallelizing existing applications in a distributed heterogeneous environment," in *4th Heterogeneous Computing Workshop*, 1995, pp. 93–100.

[97] L. Wang, H.J. Siegel, V.R. Roychowdhury, and A.A. Maciejewski, "Task matching and scheduling in heterogeneous computing environments using a genetic-algorithm-based approach," *Journal of Parallel and Distributed Computing*, vol. 47, no. 1, pp. 8–22, 1997.

[98] E.S.H.Hou, N. Ansari, and H. Ren, "A genetic algorithm for multiprocessor scheduling," *IEEE Transactions on Parallel Distributed System*, vol. 5, no. 2, pp. 113–120, 1994.

[99] A.S. Wu, H. Yu, S. Jin, K.C. Lin, and G. Schiavone, "An incremental genetic algorithm approach to multiprocessor scheduling," *IEEE Transactions on Parallel Distributed Systems*, vol. 15, no. 9, pp. 824–834, 2004.

[100] Thomas A. Feo and Mauricio G.C. Resende, "Greedy randomized adaptive search procedures," *Journal of Global Optimization*, vol. 6, pp. 109–133, 1995.

[101] H. Singh and A. Youssef, "Mapping and scheduling heterogeneous task graphs using genetic algorithms," in *5th IEEE Heterogeneous Computing Workshop*, 1996, p. 8697.

[102] J. Blythe, S. Jain, E. Deelman, Y. Gil, K. Vahi, A. Mandal, and K. Kennedy, "Task scheduling strategies for workflow-based applications in grids," in *5th IEEE International Symposium on Cluster Computing and the Grid*, 2005, pp. 759–767.

[103] J.L. Ribeiro Filho, P.C. Treleaven, and P.D. Milano, "Genetic algorithm programming environments," *IEEE Computer*, vol. 27, pp. 28–43, 1994.

[104] M. Srinivas and L.M. Patnaik, "Genetic algorithms: A survey," *Computer*, vol. 27, no. 6, pp. 17–26, 1994.

[105] T.D. Braun, H.J. Siegel, N. Beck, L.L. Boloni, A.I. Reuther, M.D. Theys, B. Yao, R.F. Freund, M. Maheswaran, J.P. Robertson, and D. Hensgen, "A comparison study of static mapping heuris-

tics for a class of meta-tasks on heterogeneous computing systems," *Heterogeneous Computing Workshop*, vol. 0, pp. 15, 1999.

[106] D. Garlan, "Research directions in software architecture," *ACM Computing Surveys*, vol. 27, no. 2, pp. 257–261, 1995.

[107] C. Lin and S. Lu, "Architectures of workflow management systems: A survey," Tech. Rep. TR-SWR-01-2008, Wayne State University, 2008.

[108] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and Grid computing 360-degree compared," in *IEEE Grid Computing Environments, in conjunction with IEEE/ACM Supercomputing*, 2008, pp. 1–10.

[109] D. Georgakopoulos, M.F. Hornick, and A.P. Sheth, "An overview of workflow management: From process modeling to workflow automation infrastructure," *Distributed and Parallel Databases*, vol. 3, no. 2, pp. 119–153, 1995.

[110] T. Oinn, M. Greenwood, M. J. Addis, M. N. Alpdemir, J.Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. J. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "Taverna: Lessons in creating a workflow environment for the life sciences," *Concurrency and computation: practice and experience*, vol. 18, no. 10, pp. 1067–1100, 2002.

[111] I. Altintas, O. Barney, and E. Jaeger-Frank, "Provenance collection support in the Kepler scientific workflow system," in *International Provenance and Annotation, Workshop*, 2006, pp. 118–132.

[112] E. Deelman and A. Chervenak, "Data management challenges of data-intensive scientific workflows," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, 2008, pp. 687–692.

[113] *Open Provenance Model*, http://twiki.ipaw.info/bin/view/Challenge/OPM.

[114] K. Ostrowski, K. Birman, and D. Dolev, "Extensible architecture for high-performance, scalable, reliable publish-subscribe eventing and notification," *International Journal of Web Services Research*, vol. 4, no. 4, pp. 18–58, 2007.

[115] A. Chebotko, S. Lu, X. Fei, and F. Fotouhi, "RDFProv: A relational rdf store for querying and managing scientific workflow provenance," *Data and Knowledge Engineering*, vol. 69, no. 8, pp. 836–865, 2010.

[116] A. Chebotko, X. Fei, C. Lin, S. Lu, and F. Fotouhi, "Storing and querying scientific workflow provenance metadata using an RDBMS," in *IEEE International Conference on e-Science and Grid Computing*, 2007, pp. 611–618.

[117] M. Atay, A. Chebotko, D. Liu, S. Lu, and F. Fotouhi, "Efficient schema-based XML-to-Relational data mapping," *Information Systems*, vol. 32, no. 3, pp. 458–476, 2007.

[118] A. Chebotko, M. Atay, S. Lu, and F. Fotouhi, "XML subtree reconstruction from relational storage of xml documents," *Data Knowledge and Engineering*, vol. 62, no. 2, pp. 199–218, 2007.

[119] T. Erl, *Service-oriented architecture concepts, technology and design*, Pearson Education Inc., 2005.

[120] C. Lin, S. Lu, X. Liang, J. Hua, and O. Muzik, "Cocluster analysis of thalamo-cortical fiber tracts extracted from diffusion tensor MRI," *International Journal of Data Mining and Bioinformatics*, vol. 2, no. 4, pp. 342–361, 2008.

ABSTRACT

SCIENTIFIC WORKFLOW INTEGRATION FOR SERVICES COMPUTING

by

CUI LIN

August 2010

Advisor:	Dr. Shiyong Lu
Major:	Computer Science
Degree:	Doctor of Philosophy

In recent years, significant scientific advances are increasingly achieved through complex scientific processes. As the exponential growth in computing technologies and scientific data, a scientific workflow may comprise a large number of heterogeneous scientific services and applications, provided by different organizations. These services, applications, and their associated data are usually distributed across heterogeneous computing environments. The integration and management of such scientific workflows are pushing the limits of current workflow technology. This dissertation presents an integrated solution to composing, scheduling, executing and developing scientific workflows and scientific workflow management systems.

To provide a foundation for workflow composition, scheduling, execution and management, we propose the first reference architecture for scientific workflow management systems. The reference architecture not only provides a high-level organization of subsystems and their interactions in a workflow system, but also provides a basis for comparison between different systems and a guidance for the architectural design of an SWFMS in a specific scientific domain. To integrate heterogeneous services and applications and enable them composed to workflows, we propose a task template model which not only provides an appropriate abstraction of heterogeneous services and applications, but also encapsulates the composition and mapping of shims and functional task components within a task interface. Our proposed task specification language (TSL) not only integrates heterogeneous services and applications into uniform workflow tasks, but also provides a solution to address both TYPE-I and TYPE-II shimming problems in composing scientific workflows. To schedule scientific workflows in emerg-

ing services computing environments, we propose two workflow scheduling algorithms, the Scalable-Heterogeneous-Earliest-Finish-Time (SHEFT) algorithm and the Scalable-Critical-Path-On-a-Resource (SCPOR) algorithm, to prioritize tasks in a workflow, map tasks onto suitable resources and order the execution of tasks on the assigned resources, so that the workflow makespan can be minimized. Our extensive experiments have shown that our proposed algorithms not only outperform other algorithms for data-intensive and compute intensive workflows, but also allow the assigned resources elastically change on demand of the scalability of workflows. To execute workflows on distributed computing environments, we propose a task run model to model the run-time behaviors of tasks. The proposed task run description language (TRDL) enables the execution of task instances constructed from heterogeneous services and applications. We also develop an SOA based task management subsystem to manage all task templates, task instances and task runs for the invocation and execution of various heterogeneous task components. Finally, our developed SOA based workflow management system, the VIEW system, and a VIEW based workflow application system, the FiberFlow system, validate our architectures, models, languages, and algorithms.

AUTOBIOGRAPHICAL STATEMENT

Cui Lin

Ms. Cui Lin is currently a PhD candidate in the Department of Computer Science at Wayne State University. She is a member of the Scientific Workflow Research Laboratory (SWR Lab), one of the top five leading scientific workflow research groups in North America. Ms. Lin was an IBM certified DB2 Database Expert, and she worked for IBM (IBM Global Services) as a system analyst and project manager after she received her BE in Computer Science from Beijing Information Science and Technology University. Her research interests include Scientific Workflows, Services Computing, Cloud Computing and Bioinformatics. She has published several refereed international journals and conference papers. Ms. Lin is a member of IEEE.