1-1-2011

A scientific workflow framework for scientific data querying and processing

Xubo Fei
Wayne State University,

A SCIENTIFIC WORKFLOW FRAMEWORK FOR SCIENTIFIC DATA QUERYING AND PROCESSING

by

XUBO FEI

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

2011

MAJOR: COMPUTER SCIENCE

Approved by:

_____

Advisor Date

_____

_____

_____

DEDICATION

To my parents with love

ACKNOWLEDGMENTS

First and foremost, I would like to thank my advisor, Dr. Shiyong Lu, for helping me finish my thesis. In my time under his advisorship, I have learned skills that I had struggled with all my life: how to write and present clearly; how to conquer large and complex problems; and how to formalize a mathematical model. He often challenged me for creative ideas and gave me a lot of insightful comments and unending encouragement. Dr. Lu's supervision in my study and research shifted my fundamental perspective from viewing computer science as a useful tool to viewing it as a world of precise logic, subtle complexity, and artistic design.

Many thanks to Dr. Jeffrey Ram from the Physiology department. I collaborated with Dr. Ram on the TangoInSilico project for about 2 years. We worked closely and held a regular meeting every week. I am always impressed by his profound knowledge, great intelligence and passion for research. I really enjoy our discussions and have learned a lot of biology knowledge and scientific methodologies during our collaboration.

I am very grateful to Dr. Shiyong Lu, Dr. Farshad Fotouhi, Dr. Jeffrey Ram, and Dr. Chandan Reddy, for serving on my dissertation committee and providing profuse encouragement and productive advice on my dissertation.

I would also thank every member of the Scientific Workflow Research Laboratory, Dr. Yi Lu, Dr. Mustafa Atay, Dr. Jamal Ali Alhiyafi, Dr. Seunghan Chang, Dr. Artem Chebotko, Dr. Cui Lin, Chunhyeok Lim, Dong Ruan, and Sha Liu, for their sincere friendship and great help.

Finally, I would like to thank my parents, Honglie Fei and Jianli Xu, for their love, support, and encouragement throughout my whole life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In recent decades, computational technologies have played an essential role in modern scientific research. While the couple between scientist and computer makes significant progress, it also creates new challenges. On the one hand, scientists increasingly rely on information and computation technologies to enable and accelerate scientific discoveries. High-performance computing such as supercomputers, clusters and grids have been popularized in many scientific laboratories [9] [10] [31]. On the other hand, computer simulation has become a popular tool for scientists from many disciplines to explore domains that are inaccessible or extremely expensive for real experiments such as the exploring evolution of the universe [112] [28], predicting global climate change [40] [2], and numerous "in silico" simulation of biological processes [109] [54]. Moreover, scientific instruments, computations and computer simulations are creating vast data stores. Researchers in many areas of science, especially in astrophysics, physics, climatology and biology, are now facing tremendous increases in data volumes, which have exceeds our capacity to store and analyze the data.

Scientists demand better frameworks to support the new generation scientific research cycle from data capture, data curation to data analysis and data visualization [67]. The increasingly availability of massive volumes of scientific data and corresponding analysis tools requires an integrated system to manage the data, the applications that analyze the data, as well as the whole scientific discovery process. A recent science article, titled "Beyond the Data Deluge" [25], concluded that, *"In the future, the rapidity with which any given discipline advances is likely to depend on how well the community acquires the necessary expertise in database, workflow management, visualization, and cloud computing technologies."*

1.1 Scientific Workflows and Scientific Workflow Management Systems

Workflow in general refers to the "automation of a business process, in whole or part, during which documents, information or tasks are passed from one participant to another for action, according to a set of procedural rules" [68]. The workflow concept evolved from the notion of process in manufacturing and the office and have been developed in the business world, as so called *business workflow*, to providing computerized facilitation and automation of business processes, including the assessment, analysis, modeling, definition and subsequent operational implementation of the core business processes of an organization.

As the computational "e-science" component of scientific research becomes more and more extensive and complex, a systematic architecture to manage various computational processes and large amount of data becomes more and more important [116]. Workflow concepts have recently been applied to organize scientific computations, so called *scientific workflows*. A *scientific workflow* is a formal specification of a scientific process, which represents, streamlines, and automates the analytical and computational steps that a scientist needs to go through from dataset selection and integration, computation and analysis, to final data product presentation and visualization. Scientific workflows share many features of business workflows, but also go beyond them. One of the main differences between scientific workflows and business workflows is that scientific workflows are more concerned with the throughput of data through various stages of programs and applications while the business workflows focus on correct, timely and secure execution of business logic. Therefore, scientific workflows are usually data-driven in that the tasks are orchestrated mostly by dataflows rather than traditional control flows. In a scientific workflow, each task has several input and output ports. The input ports receive tokens from a predecessor component(such as a task or workflow input); the outputs ports send tokens to successor components(such as a task or workflow output). In data flow perspective, Receipt of all the input data tokens will trigger the task. When the task is complete, it will then generates data tokens and send them to related output ports. Another difference is that scientific workflows are usually dynamic with highly

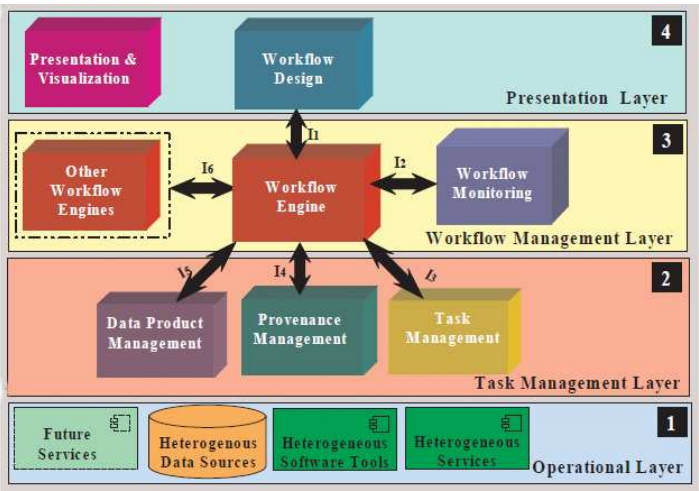


Figure 1.1: A Reference Architecture for SWFMSs.

user interactions while business workflows are static. Complex scientific experiments often involve many parameters which will be changed frequently by the domain scientists in order to refine the model. And moreover, the workflow itself is usually changed very often during the exploratory research. Therefore scientists require higher level tools with friendly working environment, which enables them to plug together problem solving components to prove a scientific hypothesis. Business workflow tools look more like traditional programming languages, and are at the wrong level of abstraction for scientists to take advantage of. Instead, scientific workflow systems are trying to provide an environment to aid the scientific discovery process through the combination of scientific data management, analysis, simulation and provenance.

A scientific workflow management system (SWFMS) is a system that supports the specification, modification, execution, failure handling, and monitoring of a scientific workflow using the workflow logic to control the order of executing workflow tasks. SWFMS has become a fundamental instrument for current and future scientific research and collaboration, which provides rich support for scientists to describe experiments, analyze data, share descriptions and results with colleagues, as well as automate the recording of vast amounts of

data products and provenance information. While a business workflow management systems (BWFMSs) focus on the management, coordination, and verification of business processes, SWFMSs focus on supporting data intensive and computation intensive scientific research projects. Figure 1.1 illustrate a reference architecture for SWFMSs proposed in [85].

1.2 Scientific Data Management

Scientific data management is one of the greatest challenges in the coming data intensive science paradigm, not only in terms of the volume, but also in terms of the heterogeneity and distributive organization. While the relational data model [37] and SQL have become standards in the commercial world, none of the existing data model alone has critical mass in the scientific community and different data models and representations exist even in the same domain. Although much science data is in the form of numeric arrays and tables, relational databases are not well accepted by scientists because the relational model lacks of some common scientific data types and SQL cannot support complex scientific computations. Several simple and convenient data models have emerged to represent arrays, tables and relationships among them, such as HDF [5], NetCDF [8] and FITS [4]. A standardization is yet to be proposed.

Recently, the coming data deluge generated by scientific instruments and simulations poses new requirements for scientific data management. Data volumes are doubling every year and many nowadays datasets can easily reach terabyte or even pegabyte level. New techniques are needed to analyze and organize the data. Moreover, the increasingly used distributed high performance computing such as Grid computing and Cloud computing often involve distributed and hierarchically organized data sets. However, current data model in Grid computing and Cloud computing are mainly file oriented and loosely organized. The current data management in such systems often relies on hard coded programs or even requires manual operations from the users.

1.3 Research Challenges

Scientific workflows are proving to be the preferred vehicle for computational knowledge extraction at a large scale. However, the research on scientific workflows is still in its infancy. This dissertation explores formal methodologies to modeling scientific workflows. Specifically, the goal of this dissertation is to address the following challenges.

How to define a dataflow-based scientific workflow composition model to support scientific workflow compositions. Scientific workflows have become a new paradigm for scientists to integrate, structure, and orchestrate heterogeneous and distributed services and applications into scientific processes to enable and accelerate many scientific discoveries. In contrast to business workflows, which focus on the modeling of controlflow oriented business processes, scientific workflows aim to model often large-scale data-intensive and computation-intensive scientific processes. This poses new exciting challenges for the management of scientific workflows [85].

We argue that there is a great need to design and implement a dataflow-based scientific workflow composition model. First, as more and more scientific research projects use scientific workflow as an enabling technology to automate and speed up the scientific discovery process, productive workflow composition that promotes workflow sharing and reuse becomes increasingly important. Second, while the goal of business workflows is to reduce human resources (and other costs) and increase revenue, 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. As a result, while business workflows are typically controlflow oriented, scientific workflows tend to be dataflow oriented. Therefore, instead of using an existing business workflow language (such as BPEL [23] and YAWL [118]), it is highly desirable to have a dataflow-based scientific workflow language to support the specification and execution of complex data-driven scientific workflows. Finally, although several dataflow-based scientific workflow languages have been implemented [89], [97], [56],

none of them provides the dataflow constructs (e.g., Map and Reduce) that are fully compositional one with another.

How to define a formal scientific data model with well-defined operators. In contrast to business data, which is usually relational and stored in databases, scientific data is often hierarchically organized and collection oriented. We argue that a scientific workflow data model should meet the following requirements. First, a scientific workflow data model should be collection oriented. Scientists often work with collection oriented datasets, such as arrays, lists, tables, or file collections, which are generated from various instruments or simulations [62]. Therefore, it is important that a scientific workflow data model can support such collection-oriented data structures. Moreover, a collection-oriented data model enables data parallelism in scientific workflows, such that multiple runs of the same workflow can be performed in parallel over collections of data. Second, a scientific workflow data model should support nested data structures. On one hand, scientific data is often hierarchically organized. For example, physiologists often classify their clinical data by different patients and dates, forming a hierarchical cluster of data. On the other hand, in scientific workflows, workflow tasks often produce lists of data products, and the execution of a workflow composed from such tasks can create increasingly nested data collections [91]. Finally, a scientific workflow data model should provide well-defined operators and their arbitrary compositions to manipulate and query scientific data collections. Such operators can become the basis for a higher-level declarative workflow language and provide a mathematical foundation for query and workflow optimization. Although several collection oriented data structures have been proposed for SWFMSs [117] [91] [132] [32], a formal data model with a set of well-defined operators is still missing.

How to design and implement a scientific workflow management system to integrate proposed techniques.

A scientific workflow management system aims to provide a framework to support the whole cycle of scientific research. To realize and implement proposed techniques into an

integrated SWFMS remains a big challenge. First, while advanced computer science techniques enabled and accelerated many scientific discoveries, they also bring burden to domain scientist who are forced to learn computer science technologies. SWFMSs are designed to provide a higher level programming abstraction and rid scientists of complicated technical details,so that they can concentrate on the research problem. Therefore, SWFMSs should provide a simple and friendly user interface and detailed techniques should be hidden inside the backstage. Second, modern SWFMSs often consist of several subsystems either loosely coupled or tightly coupled, and each implements some partial functionalities. Therefore, it is very important to maintain the consistency between subsystems, such as data typing, and system status. Finally, the coordination and communication between subsystems need to be clearly identified, including the functional interface, state transition, and the data and message interchange protocols.

1.4 Contributions

Contributions of this dissertation are as follows:

- **A Dataflow-based Scientific Workflow Composition Model.** We identified seven key requirements for a scientific workflow composition model based on a comprehensive literature review and our experience in developing the VIEW system. based on those requirements, we proposed a dataflow based scientific workflow composition model consisting of: i) a dataflow-based scientific workflow model that separates the declaration of the workflow interface from the definition of its functional body; ii) a set of dataflow constructs, including Map, Reduce, Tree, Loop, Conditional, and Curry, which are fully compositional one with another; iii) a dataflow based exception handling approach to support hierarchical exception propagation and user-defined exception handling. Our workflow composition framework is unique in that workflows are the only operands for composition; in this way, our approach elegantly solves the two-world problem in

existing composition frameworks, in which composition needs to deal with both the world of tasks and the world of workflows.

- **A Collectional Data Model.** We formalized a collection-oriented data model, called *collectional data model*, to model hierarchical collection-oriented scientific data. The new collectional model naturally extends the relational model to support hierarchical scientific data. We also proposed a set of well-defined operators to manipulate and query such data including *union* and *set difference*, *selection*, *projection*, *Cartesian product* and *renaming*. The proposed collectional operators can be composed arbitrarily to form more complex operations and the result will always be a collection.

- **An Integrated Scientific Workflow Management System.** We designed and implemented a prototypical scientific workflow management system, call VIEW. The VIEW system comprises six loosely coupled subsystems implementing our proposed techniques: a workbench to visually design and compose workflows and visualize data products, a workflow engine realizing our proposed model to execute workflows, a task manager to manage and execute heterogeneous tasks, a data product manager to store and manage scientific data products based on the collectional data model, a workflow monitor to display system status and track exceptions, and a provenance manager to store and query workflow provenance.

1.5 Roadmap

The remaining chapters of the dissertation are organized as follows. Chapter 2 reviews related research that covers the state of the art technologies of scientific workflow modeling, scientific data modeling, and scientific workflow systems. Chapter 3 proposes a new dataflow-based scientific workflow model and a set of workflow constructs to enable arbitrary hierarchical workflow compositions. Chapter 4 formalizes a collectional data model to support hierarchical collection-oriented scientific data, and a set of operators to manipulate and query

such data. Chapter 5 presents the detailed design and implementation of the VIEW system, which integrate all the proposed techniques. Finally, Chapter 6 concludes the dissertation and outlines some future research work.

CHAPTER 2

RELATED WORK

Scientific workflow has become an increasingly popular paradigm for scientific data querying and processing. As a multi-disciplinary research area, scientific workflows involve technologies from various domains. This chapter presents existing workflow and data management technologies that are pertinent to this thesis. Section 2.1 provides an overview of business workflow and scientific workflow research with a focus on models and languages. Section 2.2 discusses existing scientific data models. Section 2.3 further surveys existing scientific workflow management systems. Finally, Section 2.4 summaries this chapter.

2.1 Workflow Modeling

Workflow technology has been successfully used in business and scientific applications for many years and numerous competing proposals have been proposed to model workflow processes from opposing companies. There are two main architectural approaches to implementing workflow: *service orchestration* and *service choreography* [24]. Service orchestration means an executable business process that may interact with both internal and external services (e.g. Web services). Services interact with each other by explicitly defined controlflow or dataflow. Orchestrations can span multiple applications and/or organizations while a central process acts as a controller to the involved services and the services themselves have no knowledge of their involvement in a higher level application. BPEL [23] [12] and YAWL [118] are two representative business workflow languages that are widely adopted by the community for defining processes that can be executed on an orchestration engine. Most current scientific workflow languages including MoML [89], Xsculf [15], and our to be proposed language SWL are also based on a service orchestration model. Service choreography

focuses more on a collaboration between a collection of services in nature. Choreography describes interactions from a global perspective, meaning that all participating services are treated equally as a peer-to-peer fashion. Each party involved in the process describes only the part they play in the interaction and no process acts as a controller. All involved services are aware of their partners and when to invoke operations. WS-CDL [75] is a representative business workflow language in this area.

2.1.1 Business workflow modeling

Workflow technology was first adopted in the business community and has been developed for many years. The main purpose of business workflow is the automation of processing steps (activities) in order to accomplish some business process [14]. Below, we will introduce several representative business workflow languages.

Business Process Execution Language for Web services (or BPEL4WS) [23] [12] [76] has gained broad acceptance in industry and research. It is an XML-based language as the formal specification of business processes and business interaction protocols. BPEL4WS extends the Web services interaction model [124] and enables both, the composition of Web services and rendering the composition itself as Web services [96]. BPEL provides control constructs including $< sequence >$, $< flow >$, $< switch >$, $< pick >$ and $< while >$. BPEL also provides *control links*, together with the associated notions of join condition and transition condition, to support the definition of task precedence, synchronization and conditional dependencies [100]. BPEL has been supported by a significant number of business workflow tools, and has also been used to structure some simple scientific workflows [19]. However, while most modern scientific workflows are data driven, BPEL does not support explicit data flow. Data in BPEL is stored in shared variables that can be accessed by activities (e.g. $< assign >$ activity). Moreover, it has been noted that standard BPEL fails to support human tasks, that is, tasks that are allocated to human actors and that require these actors to complete actions, possibly involving a physical performance. Although some extensions

to BPEL are developed such as BPEL4People [77] to support human interactions, they are designed mainly to model business activities rather than scientific experiments.

Yet Another Workflow Language(YAWL) [118] is a formal language which was originally proposed to support most of the workflow patterns [17]. Those patterns characterize the desirable properties of workflow languages from the controlflow perspective. The YAWL language is based on high-level Petri nets [119] and extends it with three main constructs, or-join, cancellation sets, and multi-instance activities, to express multi-instance activities. YAWL also introduces some other constructs, such as simple choice (xor-split), simple merge (xor-join), and multiple choice (or-split), to support workflow patterns that are not easily represented using Petri nets. YAWL is recently extended, so called newYAWL [105], offering to provide holistic support for the controlflow, data and resource perspectives, and to cover many new patterns which YAWL is unable to provide direct support for, including the partial join, transient and persistent triggers, iteration and recursion.

The XML Process Definition Language (XPDL) [16] is a format standardized by the Workflow Management Coalition (WfMC) to interchange the process design, both the graphics and the semantics of a workflow business process between different workflow products like modeling tools and workflow engines. XPDL defines an XML schema for specifying the declarative part of workflow. In XPDL, the Process Definition entity provides contextual information that applies to other entities within the process. It consists of one or more activities, each comprising a logical, self-contained unit of work to be performed by either some resource or computer application. An activity may be a subflow containing the execution of a process definition that is separately specified, or a block activity that executes an activity set, or map of activities and transitions. Activities are related to one another via flow control conditions. Each individual transition has three elementary properties, the from-activity, the to-activity and the condition under which the transition is made. XPDL also contains elements to hold graphical information such as the X and Y position of the activity nodes as well as the coordinates of points along the lines that link those nodes. This distinguishes XPDL

from BPEL as the latter one does not contain elements to represent the graphical aspects of a process diagram.

WS-Choreography Definition Language (WS-CDL) [75] is an XML-based language that describes peer-to-peer collaborative and complementary behavior of multiple participants. The major difference between WS-CDL and BPEL is that the former provides a definition of the information formats being exchanged by ALL participants, while the later provides the information formats exchanged by one participant. Thus WS-CDL provides the global message exchange between participants without a specific point of view.

Many other business workflow languages are proposed. Huang et al. proposed a policy language [71] in support of the project-oriented workflow. In their model, a project can be divided into many functional modules defined in a sub process definition, either composite activities or atomic actives, and composite activities can be divided further. Stefansen et al. proposed a *SMAll Workflow Language Based on CCS (SMAWL)* [114], which aims to reduce the amount of user-specified internal synchronization while can still provide elegant constructs for the workflow patterns [17]. Gregory et al. proposed *Workflow Prolog* [63], which leverages the properties of Prolog such as its familiarity and efficiency, and allows workflow systems to be implemented in a novel declarative style. Han et al. proposed an *Ubiquitous Workflow Description Language (uWDL)* [64], to support adaptive services and specify context information on the transition constraints. Charfi et al. introduces a new unit, called aspect, to modularize crosscutting concerns in complex systems and proposed an aspect-oriented workflow language, called *AO4BPEL* [33]. Handl proposed *HotFLow* [65] for the B2B Electronic commerce project MALL2000, which is a visual language for controlling the dynamic workflow of negotiating and contracting. Wirtz introduced the *Object Coordination Nets (OCoN)* [127] approach which carries the benefits of visual software engineering techniques to the workflow area. Wong et al. proposed a process-algebraic approach [128] to model workflows as CSP processes and support various controlflow patterns. Ontology [120] techniques are also introduced to model workflows. *OWL-S* [90] provides *Split+Join* for

the parallel execution of semantic Web services while the *Web Service Modeling Ontology (WSMO)* [103] also supports parallel workflows through a set of controlflow-based transition rules which are executed in parallel.

All the above business workflow models and languages are driven by controlflows because business workflows are driven by business rules and it is important to maintain the state of a business process and to provide controlflow constructs to formulate state-based business rules. Although some constructs, such as *ForEach*, *If*, *While* in BPEL 2.0 [12]; *MultipleInstance*, *Structured Loop*, *Multiple Choice* and *Parallel Split* in YAWL [118], have been proposed for business workflows to support iteration and concurrency, they cannot be directly applied to a dataflow-based scientific workflow composition framework due to the fundamental differences between controlflow and dataflow. For example, in contrast to our to be proposed Map construct, which returns a list of results, the *ForEach* construct returns nothing (since it is a controlflow construct). Considering the dataflow-oriented nature, the Map construct is more natural for scientific workflows as the results can be directly fed to the input of subsequent workflows or tasks.

Recently, data-centric approaches have received much recognition to model medium or large sized business workflows. IBM introduced an artifact-centric approach [57] [47], which focuses on recording "business artifacts" including business objects, their life cycles, and provenance information. E-BioFlow [123], a workflow system built on top on YAWL [118], provides three perspectives (controlflow, dataflow, and resource) to support workflow design. The information of the three perspectives will all be translated to controlflows during runtime. However, in essence, these approaches are still controlflow based rather than dataflow based.

2.1.2 Scientific workflow modeling

Scientific workflow shares many similarities with business workflow, but also go beyond it [44]. There are significant discussions about the similiarities and differences between scientific workflows and business workflows [115] [121] [19] [111]. First of all, scientific workflows are more concerned with the throughput of data through various stages of programs and applications while the business workflows focus on correct, timely and secure execution of business logic. Therefore, scientific workflows are usually data-driven in that the tasks are orchestrated mostly by dataflows rather than traditional controlflows. In a scientific workflow, each task has several input and output ports. The input ports receive tokens from a predecessor component (such as a task or workflow input); the outputs ports send tokens to successor components (such as a task or workflow output). In dataflow perspective, Receipt of all the input data tokens will trigger the task. When the task is complete, it will then generates data tokens and send them to related output ports. Second, scientific workflows are usually dynamic with intensive user interactions while business workflows are static. Complex scientific experiments often involve many parameters which will be changed frequently by the domain scientists in order to refine the model. And more over, the workflow itself is usually changed very often during the exploratory research. Scientists require higher level tools with friendly visual working environment, which enables them to plug together problem solving components to prove a scientific hypothesis. Finally, while business workflows are mainly dealing with Web services coordinated via simple messages, scientific workflows often involve heterogeneous and distributed computation resources in order to process huge and complex scientific data.

As scientific workflows are more dataflow driven, we briefly review the literature of dataflow languages [73]. The name dataflow comes from the conceptual notion that a program is a directed graph and that data flows along its arcs between instructions (components). Many developments have taken place within dataflow programming languages in the past decade. The *Textual Data-Flow Language (TDFL)* [125] is one of the first purpose-built

dataflow languages. It was designed to be compiled into a dataflow graph with data streams in a relatively straightforward way. TDFL consists of a series of concepts including modules, analogous to procedures in other languages. Each module is made up of a series of statements such as assignments, conditional statements, or a call to another module. Iteration was not provided directly. *LAU* [58] is a single assignment language which includes conditional branching and loops that were compatible with this rule. It was one of the few dataflow languages that provided explicit parallelism. *Cantata* [102] is a coarse-grained visual dataflow language in which nodes contain entire functions (similarly in workflows), rather than just a primitive operation. Each input is designated a name by the programmer, who also specifies either a loop variable and bounds, or a WHILE-condition, using the names. Much features and principles in dataflow research has been inherited in scientific workflows. As a matter of fact, most scientific workflow models are typically dataflow based. However, scientific workflows are specifically designed to facilitate scientists for scientific data processing, therefore, they are usually more course-grained and also provide support for modern super computing techniques.

Many modern scientific workflows originate from Grid applications. Grid workflows have been proposed to enhance cyberinfrastructure for a wide range of scientific domains. Grid computing satisfies high-performance requirement of the complex scientific applications and enables resource sharing between collaborating organizations. Grid workflows provide an integrated and user friendly environment for domain scientists to utilize the advantages of grid computing. Pegasus [45] aims to take advantage of Grids for parallel processing at the task level and its workflow language *DAX* [3] can describe controlflow-based sequential and parallel workflows. The DAX language use the notion of $< job >$ to denote a task and use $< child >$ to define the control-flow dependencies between jobs. DAX does not explicitly support dataflow. Instead, data is transferred as parameters or files. The Swift [132] system defines the proprietary *Virtual Data Language (VDL)* [55]. VDL uses

a C-like syntax to represent XML Schema types and procedures. It enables the programmers to describe the types of both datasets (including file system data) and workflow components. It also supports the invocation of remote procedure calls to perform computations on those data objects and provides an implicitly parallel, functional programming model based on dataflow concepts. ASKALON [49] proposed an *Abstract Grid Workflow Language (AGWL)* [50] [51]. AGWL is an XML-based language designed specifically for describing Grid applications at a high level abstraction, called *activities*, without dealing with implementation details. AGWL includes the most essential workflow constructs including activities, sequence of activities, sub-activities, controlflow mechanisms, dataflow mechanisms, data repositories, and some grid workflow constructs such as parallel activities, parallel loops with pre- and post-conditions, synchronization mechanism, and event based selection of activities. There are many other proposals include *JXPL* [72] for the GridNexus [29] system, *DPML* for the DiscoveryNet system [41], *GWorkflowDL* [22] based on High-Level Petri Nets, *GPEL* [121] [122] and *GSWEL* [129] extended from *BPEL4WS*, *SWFL* [69] and *MPFL* [70] extended from *WSFL* [80] (an XML language developed by IBM for the description of Web Services compositions as part of a business process definition). [78] [130] survey and the Grid programming environments and representative Grid workflow systems.

While grid workflows provide a high level abstraction on top of the distributed Grid resources, they are limited for Grid applications and lack the ability to manage scientific data, and to utilize heterogeneous resources. Recently, several general scientific workflow models and systems are developed. Below, we review several most representative proposals.

Kepler [89] inherits the actor-oriented modeling design [26] from the Ptolemy II system [30]. Actor-oriented modeling clearly separates two modeling concerns: component communication (dataflow) and overall workflow coordination (orchestration).A scientific workflow is modeled as a composition of independent components, called actors. Actors are reusable independent blocks of computation. They consume data from a set of input ports and write data to a set of output ports. The interaction between the actors is defined by a Model of

Computation (MoC) [60]. The MoC specifies the communication semantics among ports and the flow of control and data among actors. *Directors* are responsible for implementing particular MoCs, and thus define the orchestration semantics for workflows. A variety of models of computation are supported in Kepler, including: Process Networks (PN) for pipelined current execution, Dataflow (DDF and SDF) for dataflow based execution, Continuous Time (CT) for time based execution, and Finite State Machines (FSM) and Modal Models for state based execution. Kepler also inherits Ptolemy's own *Modeling Markup Language (MoML)* [79].

Taverna [97] implements its own *XML Simple Conceptual Unified Flow Language (Xscufl)* [15]. A Taverna workflow consists of a collection of processors with both data and control links among them. A control link establishes a control dependency indicates that a processor can only begin its execution after some other processor has successfully completed its execution. Taverna [97] provides implicit iteration by allowing a user to specify the iteration strategy of each processor (Taverna's term of workflow task). Taverna can simulate control links using data links [117], and If-Else behavior can be supported by using control links and two distinguished processor called "Fail-if-false" and "Fail-if-true". Recently, a successor of Taverna has been developed, called Taverna 2 [94]. Taverna 2 implements a new model [113], which improves the original model in two main ways: (i) support for data streaming, through pipelined execution of workflows; and (ii) support for extensibility of the set of workflow operators by wrapping each processor P with a stack of execution layers such as *Loop* for iterative execution, *Branch* for conditional execution, and *Bounce*, *Failover*, and *Retry* for exception handling.

Vistrail [32] features an action-based mechanism to automatically capture workflow evolution provenance - all the trial-and-error steps follow to construct a set of data products. In Vistrail, a workflow is represented by a sequence of actions, so called a *vistrail*. A *vistrail* is essentially a tree in which each node corresponds to a version of a workflow, and the lines between the parent nodes and their children represent the actions applied to parent nodes to

obtain the child nodes. In this case, it allows scientists to explore visualizations by returning to and modifying previous versions of a workflow.

Triana [35] provides a clear separation between the abstract workflow model and the concrete task model. A *component* in Triana is the unit of execution, Components are Java classes with an identifying name, input and output ports, a number of optional name/value parameters, and a single process method. Components can also be written in other languages with appropriate wrapping code. Each component has a definition encoded in XML with a similar format to WSDL [11], which specifies the name, input and output specifications and parameters. Triana uses both dataflow and controlflow for component execution but does not provide any explicit control constructs. Instead, Loops and conditional branching in Triana are handled by specific components, i.e. a specific loop component that controls repeated execution over a sub-workflow and a logical component that controls workflow branching.

None of the existing scientific workflow models provides the constructs that are composable and can be applied on arbitrary workflows. For example, Kepler [89] provides an *IteratorOverArray* actor (Kepler's term of workflow task) to support iterated execution. However, this actor does not directly support parallel execution of its contained actor. A recently proposed scientific workflow language, Martlet [61], provides the map and fold constructs to support MapReduce-style workflows. However, because it is controlflow-based, the constructs introduced in Martlet are inapplicable to dataflow-based scientific workflows in which input ports and output ports are well-defined. Moreover, the composability of Martlet is very limited as Martlet constructs cannot be applied in a nested way. Similarly, MOTEUR [59] supports both the parallel processing of independent data with a single service on different computing resources (called "data parallelism") and parallel execution of different services with different datasets (called "services parallelism"). However, arbitrary composition of constructs is still not supported.

2.2 Scientific Workflow Data Models

Business workflows are mainly dealing with two data models: the relational data model [37] [39] and the XML data model [13]. Business data, such as financial records, medical records, personal information and manufacturing and logistical data, are usually relational and stored in relational databases. A relation is defined as a set of tuples that have the same attributes. A tuple usually represents an object and information about that object. A relation is usually described as a table, which is organized into rows and columns. All the data referenced by an attribute are in the same domain and conform to the same constraints. The relational model offers an abstracted view of data. It basically abstracts the physical structure of data storage, from the logical structure of data, and provides a set of algebraic to query and manipulate relations. It also offers a declarative interface (relational calculus) for the specification of data manipulation, which is proved to be equivalent relational algebra with [38]. The relational model is realized in a Structured Query Language (SQL) [7] and implemented in a variety of relational database management systems including Oracle, MySQL, and MS SQL Server. Business workflows are also standardized with the XML data model to transfer data between businesses processes. XML(eXtensible Markup Language) is a markup language for document containing structured information. Documents refer not only to traditional documents, but also XML data formats such as e-commerce transactions, objects, and thousands of other kinds of structured information. Since XML data is self-describing, XML is considered a means to represent semi-structured data. The basic construct of an XML document is the *element*. Elements can contain subelements. The content of an element is delimited by special markups known as *start tag* and *end tag*. The start tag is the name of the element in angle brackets; the end tag adds an extra slash character before the name. XML is a semi-structured model and provides a flexible format for data exchange between different types of databases. However, in XML, queries cannot be made as efficiently as in a more constrained structure.

As scientific workflow becomes an active research area, there is a growing interest in the development of a data model for scientific workflow management systems. Kepler [91] [48]

proposes a collection-oriented model in which a collection is a named set of heterogeneous data which can contain sub-collections to formalize a nested collection. Our collectional data model is different from Kepler's nested data collection model. On one hand, a collection in Kepler is an XML-like semistructured data structure, consisting of labeled data items, metadata items, and nested collections with possible different types and nesting levels, while our collection is structured, consisting of data items of the same type, or consisting of nested collections with the same schema and nesting levels. On the other hand, we have defined several collectional operators that generalize their relational counterparts, no such operators have been defined in the Kepler's nested data collection model. Taverna [117] adopts a list based data model, in which string is the only atomic data type and the nested list is the only data construct. Taverna provides implicit iteration to support parallel processing of a list of data products and allows a user to specify the iteration strategy on the processor to combine multiple lists with cross product or dot product. Swift [132] supports atomic data types such as integer and string, as well as a "mapped type", which maps data directly to files on disks. Swift also supports the Array structure and user-defined structures, which are similar to those used in conventional programming languages. Pegasus [45] supports File as the only data type and data operations rely on user defined tasks. VisTrails [56] supports common atomic data types including File and provides List and Tuple data structures. GridDB [87] introduces the relational model into Grid workflows by using a Set construct to cast atomic data into relations. The relational operators can then be introduced into workflows as primitive tasks. However, GridDB does not support hierarchical data collection.

Google MapReduce [42] adopts a simple data model which is a collection of key-value pairs. However, this model does not support nested collections. Pig Latin [99] proposes a nested data model in which tuples are basic building blocks. Pig Latin provides the Bag structure to construct collections of tuples and the Map structure to construct collections of key-value pairs where the values can be of any data types. The schemas of Bag and Map are loose in that data items within one collection can be of different types. Pig Latin does

not provide operators except for basic storage and retrieval. DryadLINQ [131] adopts the LINQ data model consisting of strongly-typed collections of .NET objects. LINQ supports data collections including the dictionary data structure which contains key-value pairs and provides SQL-like operators. However, nested dictionary structure is not supported so far.

2.3 Scientific Workflow Management Systems

Business workflow management systems (BWFMSs) originate from office automation systems about four decades ago, and grow fast during the last two decades in industry. Many business workflow management systems have been developed to orchestrate and coordinate business processes. For example, the YAWL system [118] [105] is developed on a service-oriented architecture and consists of four YAWL services: YAWL worklist handler to assign work to users of the system so that users can accept work items and signal their completion ; YAWL web services broker to discover services, YAWL interoperability broker interconnect different workflow engines, and custom YAWL services connects the engine with an entity in the environment of the system. Some other systems include [66] [107] [21] [93] [81] [92].

Scientific workflow managements systems (SWFMSs) emerge in recent years in order to provide an integrated platform for facilitate scientists to design workflow, monitor workflow execution, visualize data product, and query provenance. Comparing to BWFMSs, research and development of SWFMSs are still in their infancy. Until recently, a reference architecture [85] was proposed, which clearly defined the responsibility of a SWFMS, and clarified functionalities. Most SWFMSs haven been mentioned in Section 2.1 from the aspect of modeling and language, in section we will review their system design and implementation.

Kepler [89] inherited the Ptolemy II system [30], which is to tightly coupled system include a user interface to design workflows and an engine to execute workflows. Kepler's strength include its mature library of actors, which are mainly local application for biology, ecology, geology, astrophysics and chemistry, and its suite of directors that provide flexible control strategies for the composition of actors. The Kepler system also implements a

novel hybrid type system for modeling scientific data that separates structural data types and semantic data types [26]. The well defined data type systems can facilitate the design and implementation of workflows by constraining the possible values and interpretations of data in a scientific workflow.

Taverna [97] [98] focuses particularly on orchestration of applications and services in the bioinformatics domain. Taverna is designed in a three- tiered model for describing resources and their interoperation at different levels of abstraction: An abstract layer to present the workflow from a user view, hiding the complexity of the service interactionsa Freefluo enactor manages different services in the low level with an extensible processor plug-in architecture; and an execution layer in between to interpret internal object model that handles controlflows such as implicit iteration and fault recovery on behalf of the user.

Triana [35] was originally developed as a data analysis problem-solving environment for gravitational wave detection project. The system is designed in an two layer architecture: first, users are allowed to use compose workflows graphically by dragging programming components called units or tools onto a workspace. Components are connected by data and control links. Triana workflows will be recorded and sent to the Grid Application Prototype Interface (GAP Interface) that can execute any sub-workflow and communicate with other Triana services they are connected to. GAP provides a subset of the functionality of the GAT (Grid Application toolkit, created by GridLab [20]). The GAP is used to interface with Triana services and provides us with the middleware independent view of the underlying services and interactions across the Grid.Three bindings to GAP are currently supported in Triana: Web services, P2PS(a lightweight P2P middleware capable of advertisement, discovery and virtual communication within ad-hoc P2P networks), and Jxta (a set of protocols for P2P discovery and communication within P2P networks).

Pegasus [45] is a framework which maps scientific workflows onto distributed resources such as a Grid. Abstract workflows designed by a domain scientist are independent of any resources they will be executed on. By doing this, Pegasus leverages abstraction for workflow

description to obtain ease of use, scalability, and portability. Pegasus provides a compiler to map from high-level descriptions to executable workflows and it then use Artificial Intelligence planning techniques to find a mapping of the tasks to the available resources for execution at runtime. The execution of tasks are handled by the Condor system, which is an open source high-throughput computing software framework for coarse-grained distributed parallelization of computationally intensive tasks in Grid.

ASKALON [49] is designed in a similar architecture to Pegasus. ASKALON as allows the user to compose the Grid workflow by using a graphical user interface or writing an AGWL program directly. It then uses a transformation system to compile AGWL into a concrete representation through mapping abstract activities into specific Activity Deployments deployed in the Grid. Finally A concrete representation is interpreted by the underlying workflow runtime environment of ASKALON to construct and execute the Grid workflow application on a Grid infrastructure.

VisTrail [32] is the first system to provide support for tracking workflow evolution by maintaining detailed provenance of the exploration processłboth within and across different versions of a dataflow [56]. Users create and edit dataflows using the VisTrail Builder user interface. The dataflow specifications are saved in the VisTrail Repository and users can interact with saved dataflows by invoking them through the VisTrail Server or by importing them into the Visualization Spreadsheet, which stores all dataflow instances. The VisTrail Cache Manager keeps track of operations that are invoked and their respective parameters. Therefore, only new combinations of operations and parameters need to be executed.

2.4 Chapter Summary

This chapter has presented background information that is relevant to the rest of this thesis. This chapter was composed of three main sections. The first section introduced the background of business workflow modeling, the state-of-the-art scientific workflow modeling research, and discussed their differences. The second section continued with a review

of recent trends in scientific data management. The third section of this chapter presented several representative existing scientific workflow management systems.

CHAPTER 3

A SCIENTIFIC WORKFLOW COMPOSITION MODEL

Scientific workflows are designed to integrate and structure various local and remote heterogeneous data and service resources to perform in silico experiments to produce significant scientific discoveries. Although several scientific workflow management systems (SWFMSs) have been developed, a formal scientific workflow composition model in which workflow constructs are fully compositional one with another is still missing. In this chapter, we propose a new scientific workflow composition model. We first discuss key requirements for a scientific workflow composition model in Section 3.1. We then propose a new scientific workflow model in Section 3.2, with a set of workflow constructs in Section 3.3, including Map, Reduce, Tree, Loop, Conditional, and Curry, which are fully compositional one with another. Section 3.5 introduce a dataflow based exception handling approach. We also present two case studies in Section 3.6 to validate our proposed techniques. Section 3.7 concludes this chapter.

3.1 Key Requirements for a Scientific Workflow Composition Model

Based on a comprehensive study of the workflow literature and our own experience from the development of the VIEW system [85], we identify the following seven key requirements for a scientific workflow composition model.

R1: Programming-in-the-large. The concepts of "programming-in-the-large" and "programming-in-the-small" were first introduced by Frank DeRemer and Hans Kron in 1976 [46]. While programming-in-the-large focuses on high-level abstractions of modules and the modeling of their interactions and coordination, programming-in-the-small focuses on low-level programmatic implementation of modules and functionalities. Given the high-level orchestration

and integration nature of scientific workflow composition, a scientific workflow composition model should fall in the programming-in-the-large paradigm.

R2: Dataflow programming model. While in the imperative (controlflow-based) programming model, the order of program execution is explicitly specified by controlflow constructs, such as sequential, conditional, and loop, in the dataflow-based programming model, the availability of input data for a module initiates the execution of the module and the movement of data through modules determines the execution order of the whole program. Since most scientific workflows aim at data processing and scientific analysis problems, scientific workflow composition model should be dataflow-based. Although from a user's perspective, constructs such as Loop and If-Else are important, we show later in this section that their dataflow-based counterparts are possible. Moreover, a dataflow-based workflow model features implicit parallelism: workflow modules run in parallel by default unless there is an explicit specification that one module needs an input data that is to be produced as the output of another module. Since the dataflow-based programming model [74] eliminates the shared memory assumption and the need of program counter and control sequencer, a dataflow-based scientific workflow composition model will be able to more easily leverage the parallelism enabled by today's variety of parallel and distributed computing infrastructures (Grids, Clouds, multicore, and multiprocessor systems).

R3: Composable dataflow constructs. Current dataflow based workflow languages are usually very simple, and contain only basic data links between components. In order to address the requirements of the more and more complex e-science applications, some languages borrowed several common controlflow constructs from business workflow languages. However the semantics is thus obscured and becomes difficult to formalize because of the combination of controlflow and dataflow. Therefore we argue that composable dataflow constructs are essential for a scientific workflow composition model. In contrast with controlflow constructs, which are used to control and coordinate processes, dataflow constructs are featured with efficient and systematic data processing including: data parallelism and aggregation;

recursive data processing with finite or infinite loops; data-dependent conditional branching. Dataflow constructs should also be composable. The ability to combine basic constructs and build more complicated ones will greatly improve the expressive power of the scientific workflow composition model.

R4: Workflow encapsulation and hierarchical composition. A scientific workflow composition model should facilitate encapsulation and support hierarchical workflow composition. On one hand, one of the most import features of scientific workflows is to allow the reuse and sharing of scientific processes by workflow encapsulation [27] [110]. A scientific workflow model should provide input/ouput interface and implementation details. Such well encapsulated modules represent a separation of concerns and improve maintainability. On the other hand, a scientific workflow model should support hierarchical composition so that the users are able to compose workflows using existing scientific workflows and break down large-scale scientific workflow into smaller ones. This ability greatly improves the power of modeling of complex scientific processes and encourages scientific collaborations [88].

R5: Single-assignment property. To ease provenance tracking and workflow scheduling, a scientific workflow composition model should have the *single-assignment property*, in which data products are treated as immutable artifacts; they can be created and transported, but never updated. First, scientific discovery produced from scientific workflows must be reproducible, requiring the acyclicity of provenance graphs and the immutability of data products [95]. The violation of this property might lead to incorrect data dependencies and thus compromise reproducibility. Figure 1 illustrates an example of provenance for a workflow consisting of three tasks: T_1 takes input of d_1 and produces d_2; T_2 consumes d_2 and generate d_3; T_3 consumes d_3 and generates d_1' to replace d_1. If the single-assignment property is respected, then d_1' will be a different data product, and we can derive the acyclic dependency graph shown in Figure 3.1(a): d_1' depends on d_3; d_3 depend on d_2, and d_2 depends on d_1. However, if the single-assignment property is not enforced, then d_1' and d_1 will be treated as the same data product and will be represented by one single node, resulting in a cyclic provenance

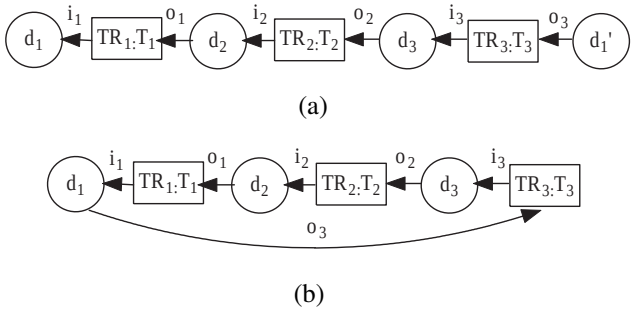


Figure 3.1: (a) Correct data dependencies under the single-assignment property; (b) incorrect data dependencies due to violation of the single-assignment property.

graph shown in Figure 3.1(b). Based on transitivity, one can infer that d_2 depends on d_3, a false data dependency relationship. Second, the single-assignment property can eliminate the interference caused by parallel access (read and write) of data products, which can result in inconsistent and undesirable intermediate or final results that would not be obtainable if workflow tasks are run in a serial fashion. Third, the single-assignment property can greatly facilitate the realization of massive parallelism: multiple workflow tasks can be started as long as their input data products become available; the single-assignment property ensures the well-defined availability time of each data product; data products can be transported to their consumers directly and removed after consumption without first being stored and then retrieved. As a result, the single-assignment property is assumed by many functional program languages and dataflow programming languages [74] [126]. Finally, unlike the transaction data in business workflows which need to be updated and changed frequently, most scientific datasets are accessed in a read-only manner and updates to datasets are usually not required [34]. Therefore, single-assignment will unlikely have negative impact on the computation and processing of scientific datasets.

R6: Physical and logical data models. Scientific applications usually involve heterogeneous and distributed data [62]. Data management is thus becoming one of the key challenges of SWFMSs [43]. We argue that scientific workflow composition model should provide both a physical data model and a logical data model, as well as the mapping between them. First,

a physical data model is important for the management of distributed data storage (such as local files, databases, and remote files) and heterogeneous physical representations (such as different formats representing the same data). Second, the logical data model provides data typing and data structures. In order to maintain the integrity and consistency of the scientific workflow composition model, a formal data typing system is required to detect "type errors". Furthermore, data structures with well-defined operators/constructs are also essential for storing and organizing collection of data tokens. Third, the separation of two data models allows the workflow users operate only on the logical data model and can be freed from physical data management [55]. As a result, changing of the underlying physical data model will breaking the scientific workflow model. Finally, an explicit and standard data mapping layer with precise metadata and explicit data access is necessary to guarantee the efficient and consistent mapping between the physical data model and the logical data model.

R7: Task level and workflow level exception handling. Exceptions in scientific workflows may happen in both the task layer and the workflow layer. A scientific workflow composition model should be able to capture and handle exceptions in both layers. First, while business workflows usually consists of Web services and exception handling in business workflows focuses on service exceptions such as service failure or deadline expiry [18], [106], [36], scientific workflows may involve heterogeneous tasks (e.g., local executables, grid applications, cloud services) and exception handling in scientific workflows is thus required to be able to detect and integrate heterogeneous exceptions generated by those tasks. Second, because scientific workflows are usually hierarchial and even distributed, exception handling in scientific workflows should also be hierarchical and exception propagation should be supported. Third, exceptions in scientific workflows are sometimes very important for scientists to detect hidden problems, improve scientific models and even achieve new scientific discoveries. Therefore, despite traditional failure handling techniques, a scientific workflow composition model should allow the users to introduce new exceptions and provide user-defined handlers.

3.2 Scientific Workflow Model

In current scientific workflow models, the workflow is defined as a composition of tasks which are either primitive or composite. Those kind of models therefore need to deal with both the world of tasks and the world of workflows. As a result, existing models cannot efficiently support workflow compositions. As shown in Figure 3.1(a), in order to create a three-level hierarchical workflow W_b using existing workflow W_c, first we need to map W_c to a composite task T_b and then use T_b to compose W_b. Similar mappings will also be needed in order to compose W_a using W_b. Those mappings between workflows and composite tasks are mathematically inelegant and lack mathematical properties to reason about workflow compositions. Inspired by functional programming, we propose a new dataflow-based scientific workflow model with a strong functional flavor. As shown in Figure 3.1(b), workflows are the only operands for composition and the two-world problem is thus avoided. In our proposed scientific workflow model, the declaration of the workflow interface is separated from the definition of its functional body. Such a separation provides an abstraction mechanism that makes it possible to introduce dataflow constructs that are fully composable one with another. Specifically, our proposed scientific workflow model consists of the following two layers:

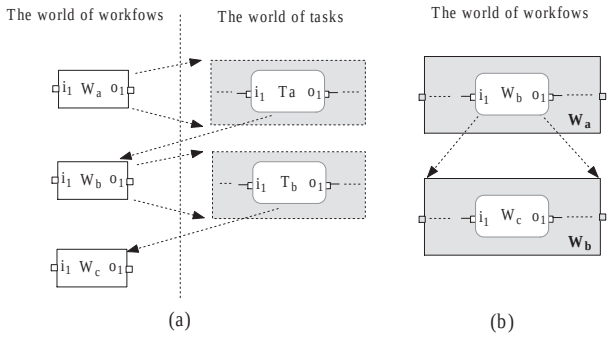


Figure 3.2: (a) Traditional scientific workflow model; (b) our proposed scientific workflow model.

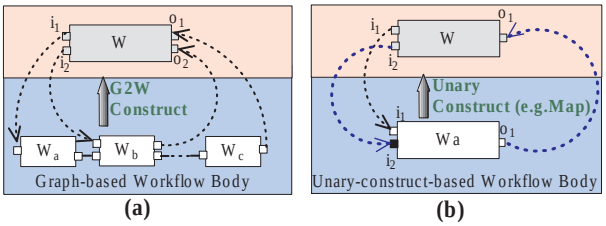


Figure 3.3: (a) a graph-based workflow; (b) a unary-construct-based workflow.

- The *logical layer* contains the *workflow interface* that models the input ports and output ports of a workflow. The details of the workflow body definition is transparent to this layer.

- The *physical layer* contains the *workflow body* that models the physical implementation of the workflow. Depending on different implementations, a workflow can be either primitive or composite. Primitive workflows are the building blocks of our model with predefined implementations while composite workflows are composed from existing workflows by dataflow constructs.

Our proposed scientific workflow model, is extensible in the sense that future dataflow constructs can be easily introduced into the model without affecting the compositionality of existing dataflow constructs. Below, we provide a brief overview of three kinds of workflows; but note that such differentiation is made only at the physical layer, not at the logical layer, thus, workflows are uniform objects in our model and they can be composed with each other using various dataflow constructs.

Primitive workflows. A set of primitive workflows serve as the basic building blocks of a scientific workflow composition framework. These primitive workflows are the abstractions of heterogeneous and distributed services and applications (tasks) that are dynamically mapped to resources at runtime. A more detailed description of this abstraction and mapping

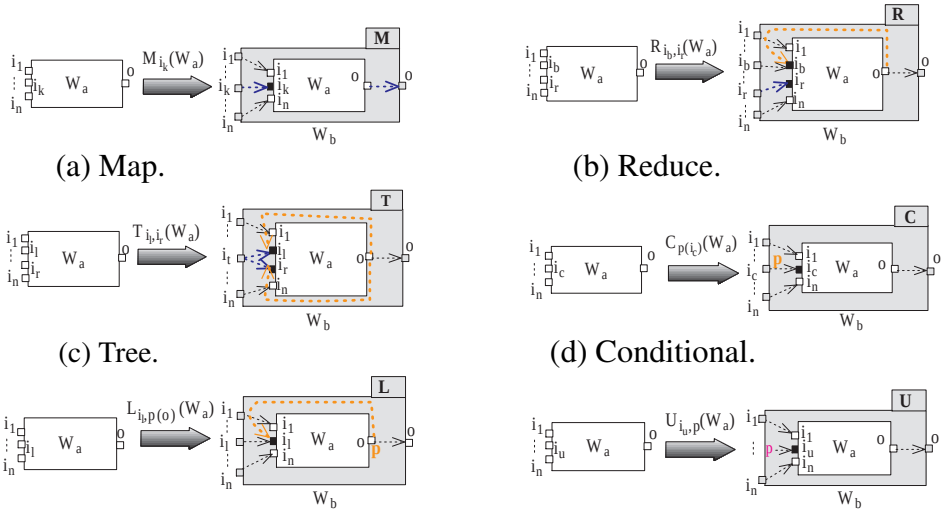
technique can be found in Chapter 5. Our proposed workflow composition framework, however, is orthogonal to how each primitive workflow is built and mapped to resources during execution.

Graph-based workflows. A set of workflows can be connected to each other via their ports through data channels to form a *workflow graph* G. During workflow execution, these workflows communicate with each other by passing data through data channels. As shown in Figure 3.3(a), the $G2W$ construct is then applied to workflow graph G to construct a graph-based workflow. $G2W$ essentially performs the mapping between the input/output ports of a workflow and the input/output ports of the workflows in its constituent workflow graph, and thus exposes some of the input/output ports of the workflows in G as the input/output ports of the target workflow.

Unary-construct-based workflows. A unary construct U can be seen as a mapping from workflows to workflows. Therefore, given a workflow W, $U(W)$ is another workflow whose behavior depends on both W and U. Unary constructs are very useful to enhance the capability of an existing workflow without coding effort and to promote the reuse of existing workflows in various contexts. For example, our to-be-proposed Map construct can be used to transform a workflow that can only process a single data product to one that can perform the parallel processing of a list of data products. As shown in Figure 3.3(b), a unary-construct essentially performs the mapping between the input/output ports of a workflow and the input/output ports of its constituent workflow, and carries out the semantics that is defined for the unary construct during runtime (e.g., the map port of the Map construct).

3.3 Scientific Workflow Constructs and Composition

A unary construct can be applied on a many-inputport-one-outputport workflow. As shown in Figure 3.4, six common unary constructs are currently supported by our model: Map, Reduce, Tree, Conditional, Loop, and Curry. More specifically, to apply a unary construct S on W_a, we define a new unary-construct-based workflow $W_b = S(W_a)$. W_b has exactly



(a) Map. (b) Reduce. (c) Tree. (d) Conditional. (e) Loop. (f) Curry.

Figure 3.4: Six unary workflow constructs.

the same number of input/output ports as W_a. Moreover, there is an isomorphic mapping between the input/output ports of W_a and W_b. Each corresponding pair of ports have the same type, i.e., $dom(W_a.i_j) = dom(W_b.i_j)$ $(j = 1, ..., n)$, and $dom(W_a.o) = dom(W_b.o)$, except for the designated ports specified by the constructs.

Although the unary constructs can only be applied to workflows with single output, it is not a limitation. The application of the construct on general workflows can be simulated by the assistance of some primitive workflows for list operations such as *Merge* and *Split* (illustrated by a later example).

In contrast to the original MapReduce model [42], which supports only task-level Map jobs (tasks), our Map and Reduce constructs can be applied to arbitrary scientific workflows; moreover, the original MapReduce model can only process key/value pairs, while our model can process data products of various types. Therefore, our model promotes the power of MapReduce from the task level to the workflow level and enables Map and Reduce fully composable with themselves and with other dataflow constructs in both flat and hierarchical manners.

3.3.1 The Map Construct

The Map construct enables the parallel processing of a list of data products based on a workflow that can only process a single data product. As illustrated in Figure 3.4(a), given a workflow $W_a([i_1, \ldots, i_n], o)$ with n input ports, i_1, i_2,..., and i_n, and one output port o, to apply the Map construct on W_a, one of the input ports of W_a, $i_k \in [i_1, \ldots, i_n]$, is designated as the *map port* which takes a list of data products that need to be processed in parallel. If $W_a.i_k$ has type T_1, then $W_b.i_k$ has type List of T_1; if $W_a.o$ has type T_2, then $W_b.o$ has type List of T_2. The semantics of the Map construct $M_{i_k}(W_a([i_1, \ldots, i_n], o))$ can be formulated by the following equation:

$$\begin{aligned} W_b.o &= W_b(i_1, \ldots, [i_{k_1}, i_{k_2}, \ldots, i_{k_m}], \ldots, i_n) \\ &= [W_a(i_1, \ldots, i_{k_1}, \ldots, i_n), \ldots, \\ &\quad W_a(i_1, \ldots, i_{k_m}, \ldots, i_n)] \end{aligned} \tag{3.1}$$

Our Map construct does not require key-value pairs as in the traditional MapReduce model. Instead, our Map construct takes a list of data products as input and the index of a data product within a list can be considered as a default "key". Our Map construct is order-preserving in the sense that each output data product has the same index as that of the corresponding input data product.

For example, Figure 3.5 illustrates a workflow W_2 that multiplies each pair of numbers in the input list. Given workflow W_1 which takes a pair of numbers as input and outputs their product, W_2 is created from W_1 by applying the Map construct with the input port i_1

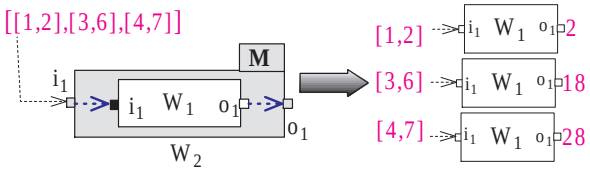


Figure 3.5: Workflow W_2 created by applying the Map construct on W_1.

designated as the map port. Given an input list $[[1,2],[3,6],[4,7]]$, the output of W_2 is:

$$\begin{aligned} W_2.o &= W_2([[1,2],[3,6],[4,7]]) \\ &= [W_1([1,2]), W_1([3,6]), W_1([4,7])] \\ &= [2, 18, 28] \end{aligned}$$

3.3.2 The Reduce Construct

The Reduce construct enables the aggregation of a list of data products to a single data product based on a workflow that aggregates *only* two of input data products. As illustrated in Figure 3.4(b), to apply the Reduce construct on a workflow $W_a([i_1,\ldots,i_n],o)$, an input port, $i_r \in [i_1,\ldots,i_n]$ is designated as the *reduce port*, which takes input from the list of data products that need to be aggregated, another input port, $i_b \in [i_1,\ldots,i_n]$ is designated as the *base port*, which takes input either from an initial base data product or from the intermediate aggregation data product that is produced as the output of the previous iteration of aggregation. If $W_a.i_r$ has type T_1, then $W_b.i_r$ has type List of T_1. Moreover, since port i_r may take input from the previous output, it is required that $dom(W_a.o) \subseteq dom(W_a.i_b)$. The semantics of the Reduce construct $R_{i_b,i_r}(W_a([i_1,\ldots,i_n],o))$ can be formulated by the following equation:

$$\begin{aligned} W_b.o &= W_b(i_1,\ldots,i_b,\ldots,[i_{r_1},i_{r_2},\ldots,i_{r_m}],\ldots,i_n) \\ &= o_m \\ where\ \ o_1 &= W_a(i_1,\ldots,i_b,\ldots,i_{r_1},\ldots,i_n) \\ o_2 &= W_a(i_1,\ldots,o_1,\ldots,i_{r_2},\ldots,i_n) \\ &\ldots \\ o_m &= W_a(i_1,\ldots,o_{m-1},\ldots,i_{r_m},\ldots,i_n) \end{aligned} \tag{3.2}$$

For example, Figure 3.6 illustrates a workflow W_3 which calculates the sum of all the numbers in the input list. W_3 is created from a predefined workflow *Add* by applying the

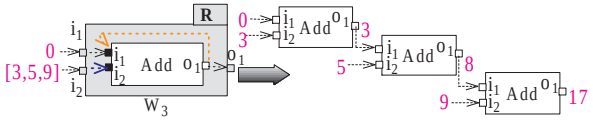


Figure 3.6: Workflow W_3 created by applying the Reduce construct on an *Add* Workflow.

Reduce construct with input ports i_1 and i_2 designated as the base port and the reduce port respectively. A default value 0 is set on the port i_1 as the base value. Given an input list $[3, 5, 9]$, the output of W_3 is:

$$o_1 = Add(0, 3) = 3$$

$$o_2 = Add(3, 5) = 8$$

$$o_3 = Add(8, 9) = 17$$

$$and \;\; W_3.o = W_3(0, [3, 5, 9])$$

$$= o_3 = 17$$

3.3.3 The Tree Construct

The Tree construct enables parallel aggregation of a list of data products. In contrast to the Reduce construct which performs a sequential aggregation, the Tree construct aggregate the list pairwisely as a binary tree until one single aggregated product is generated.

As illustrated in Figure 3.4(c), to apply the Tree construct on a workflow $W_a([i_1, i_2, \ldots, i_n], o)$, two input ports, $i_l, i_r \in [i_1, i_2, \ldots, i_n]$ are designated as the *left tree port* and the *right tree port*. The resulting unary-construct-based workflow will have a corresponding *tree port* which takes inputs of a list of data products that need to be aggregated. If $W_a.i_l$ and $W_a.i_r$ have type T_1, then $W_b.i_t$ has type List of T_1. The semantics of the Tree construct

$T_{i_l,i_r}(W_a([i_1, i_2, \ldots, i_n], o))$ can be formulated by the following recursive equation:

$$
\begin{aligned}
& W_b.o = W_b(i_1, \ldots, [i_{r_1}, \ldots, i_{r_m}], \ldots, i_n) \\
& if \;\; m = 1, \\
& W_b.o = i_{r_1} \\
& if \;\; m > 1, \\
& W_b.o = W_a(i_1, \ldots, \\
& \qquad W_b(i_1, \ldots, [i_{r_1}, \ldots, i_{r_{\lceil m/2 \rceil}}], \ldots, i_n), \\
& \qquad W_b(i_1, \ldots, [i_{r_{\lceil m/2 \rceil + 1}}, \ldots, i_{r_m}], \ldots, i_n), \\
& \qquad \ldots, i_n)
\end{aligned}
\tag{3.3}
$$

For example, Figure 3.7 illustrates a workflow W_4 which calculates the sum of the input list. W_4 is created from a primitive workflow *Add* by applying the Tree construct with input ports i_1 and i_2 designated as the left tree port and the right tree port respectively. Given an input list $[0, 3, 5, 9]$, the output of W_3 is:

$$
\begin{aligned}
W_4.o &= W_4([0, 3, 5, 9]) \\
&= Add(Add(0, 3), Add(5, 9)) = 17
\end{aligned}
$$

The application of the Tree construct will change the aggregation order and the result might be different with sequential aggregation (such as Reduce). However, below we will

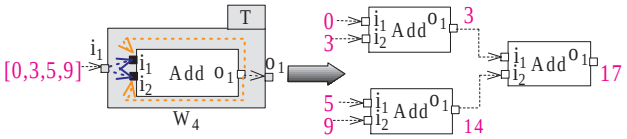


Figure 3.7: Workflow W_4 created by applying the Tree construct on an *Add* Workflow.

show that if the base workflow satisfies some property, the result of tree aggregation will be equivalent with sequential aggregation.

Definition 3.3.1 (A) workflow $W_a([i_1, \ldots, i_l, \ldots, i_r, \ldots, i_n], o)$ is *associative* with ports i_l, i_r if and only if it satisfies the *associativity law*:

$$\begin{aligned} & W_a(p_1, \ldots, W_a(p_1, \ldots, d_1, \ldots, d_2, \ldots, p_{n-2}), \ldots, d_3, \ldots, p_{n-2}) \\ = & W_a(p_1, \ldots, d_1, \ldots, W_a(p_1, \ldots, d_2, \ldots, d_3, \ldots, p_{n-2}), \ldots, p_{n-2}) \\ & for\ \ all\ \ inputs\ \ d_1, d_2, d_3\ \ and\ \ p_1, \ldots, p_{n-2} \end{aligned} \tag{3.4}$$

$\diamondsuit$

Typically, we define that a binary workflow $W_a([i_l, i_r], o)$ is *associative* if and only if it satisfies the *associativity law*:

$$\begin{aligned} W_a(W_a(d_1, d_2), d_3) = W_a(d_1, W_a(d_2, d_3)) \\ for\ \ all\ \ inputs\ \ d_1, d_2, d_3 \end{aligned} \tag{3.5}$$

Below we will show that if the base workflow is associative with the *left tree port* and the *right tree port*, given any list of inputs, the result of the Tree construct based workflow will be the same as sequential aggregation using the base workflow. We will start the proof from the binary workflows.

Theorem 3.3.2 *Given a binary workflow $W_a([i_1, i_2], o)$ that is associative, for any input list $[d_1, \ldots, d_{2^n}]$ $(n \geqslant 1)$, which can be constructed into a perfect binary tree with depth of n, the unary-construct-based workflow $W_b = T_{i_1,i_2}(W_a)$ satisfies the following equation:*

$$W_b([d_1, \ldots, d_{2^n}]) = W_a(\ldots W_a(W_a(d_1, d_2), d_3) \ldots, d_{2^n}) \tag{3.6}$$

Proof: [Proof of the Main Theorem] **Basis**: Show that the statement holds for n = 1. Obviously, we have $W_b([d_1, d_2]) = W_a([d_1, d_2])$, so the statement holds.

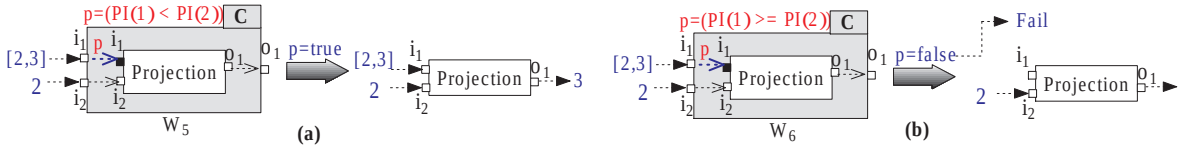


Figure 3.8: (a) W_5 created by applying the Conditional construct on the Projection workflow with a predicate $p = (PI(1) < PI(2))$; (b) W_6 created by applying the Conditional construct on the Projection workflow with an opposite predicate $p = (PI(1) >= PI(2))$.

Inductive step: Show that if the statement holds for n=k, then the statement also holds for n=k+1. This can be done as follows. Assume the equation holds for n=k, that is:

$$W_b([d_1, \ldots, d_{2^k}]) = W_a(\ldots W_a(W_a(d_1, d_2), d_3) \ldots, d_{2^k})$$

and also we can have

$$W_b([d_{2^k+1}, \ldots, d_{2^{k+1}}]) \\ = W_a(\ldots W_a(W_a(d_{2^k+1}, d_{2^k+2}), d_{2^k+3}) \ldots, d_{2^{k+1}})$$

Then for n=k+1,

$$
\begin{aligned}
& W_b([d_1, \ldots, d_{2^{k+1}}]) \\
&= W_b(W_b([d_1, \ldots, d_{2^k}]), W_b([d_{2^k+1}, \ldots, d_{2^{k+1}})]) \\
&= W_a(W_a(\ldots W_a(W_a(d_1, d_2), d_3), \ldots, d_{2^k}), \\
&\qquad W_a(\ldots W_a(W_a(d_{2^k+1}, d_{2^k+2}), d_{2^k+3}), \ldots, d_{2^{k+1}})) \\
&= W_a(W_a(W_a(\ldots W_a(W_a(d_1, d_2), d_3), \ldots, d_{2^k}), \\
&\qquad W_a(\ldots W_a(W_a(d_{2^k+1}, d_{2^k+2}), d_{2^k+3}), \ldots, d_{2^{k+1}-1})), \\
&\qquad d_{2^{k+1}}) \\
&\quad \ldots \\
&= W_a(\ldots W_a(W_a(d_1, d_2), d_3) \ldots, d_{2^{k+1}})
\end{aligned}
$$

□

Corollary 3.3.3 *Given a binary workflow $W_a([i_1, i_2], o)$ that is associative, the unary-construct-based workflow $W_b = T_{i_1,i_2}(W_a)$ satisfies the following equation for any input list $[d_1, \ldots, d_n]$:*

$$W_b([d_1, \ldots, d_n]) = W_a(\ldots W_a(W_a(d_1, d_2), d_3) \ldots, d_n) \tag{3.7}$$

Proof: Given $W_a([i_1, i_2], o)$ that is associative, we create another workflow $W_c([i_1, i_2], o)$ defined as:

$$
\begin{aligned}
W_c(d_1, d_2) = d_{id}(d_1 = d_{id}, d_2 = d_{id}) \\
\quad or \quad d_2(d_1 = d_{id}, d_2 \neq d_{id}) \\
\quad or \quad d_1(d_1 \neq d_{id}, d_2 = d_{id}) \\
\quad or \quad W_a(d_1, d_2)(d_1 \neq d_{id}, d_2 \neq d_{id})
\end{aligned}
\tag{3.8}
$$

where d_{id} is a reserved identity data product of $W_c(d_1, d_2)$ that will not be used in any other normal user inputs. We can simply prove that W_c is also *associative* by enumerating all the possibilities (due to the page limit, the proof is skipped).

Obviously, we can replace W_a with W_c without affecting the results for any inputs, then for any input lists $[d_1, \ldots, d_n]$ ($d_i \neq d_{id}$), the unary-construct-based workflow $W_d = T_{i_1,i_2}(W_c)$ satisfies the following equation:

$$\begin{aligned} W_d([d_1, \ldots, d_n]) &= (T_{i_1,i_2}(W_c))([d_1, \ldots, d_n]) \\ &= (T_{i_1,i_2}(W_a))([d_1, \ldots, d_n]) \\ &= W_b([d_1, \ldots, d_n]) \end{aligned} \tag{3.9}$$

For any input list $[d_1, \ldots, d_n]$ with length n, we will increase the length to 2^k where $2^{k-1} < n \leq 2^k$ and fill with identity data products d_{id}. By definition we can derive that

$$W_d([d_1, \ldots, d_n, d_{id}, \ldots, d_{id}]) = W_d([d_1, \ldots, d_n]) \tag{3.10}$$

By theorem 3.3.2, we have:

$$\begin{aligned} &W_d([d_1, \ldots, d_n, d_{id}, \ldots, d_{id}]) \\ &\qquad = W_c(\ldots W_c(W_c(d_1, d_2), d_3) \ldots, d_{id}) \\ &\qquad = W_a(\ldots W_a(W_a(d_1, d_2), d_3) \ldots, d_n) \end{aligned} \tag{3.11}$$

From equation 3.10 and 3.11, we have

$$W_d([d_1, \ldots, d_n]) = W_a(\ldots W_a(W_a(d_1, d_2), d_3) \ldots, d_n) \tag{3.12}$$

From equation 3.9 and 3.12, we can conclude equation 3.7.

□

Corollary 3.3.3 can be extended to multi-port workflows that are associative with two ports since inputs of other ports can be considered as arguments. We will show the proof in the later section with the help of the to-be-proposed Curry construct.

3.3.4 The Conditional Construct

The Conditional construct enables the conditional execution of a workflow based on a condition on one of the inputs.

As illustrated in Figure 3.4(d), to apply the Conditional construct on a workflow $W_a([i_1, \ldots, i_n], o)$, one of the input ports of W_a, $i_c \in [i_1, \ldots, i_n]$, is designated as the *conditional port*, on which a logical test will be calculated based on the input data product. A predicate p will be provided by the user as a parameter to evaluate the output of the *conditional port* and W_a can be executed only if p evaluates to be true. p can be modified by the user dynamically and the workflow behavior will thus be changed accordingly. The semantics of the Conditional construct $C_{p(i_c)}(W_a([i_1, \ldots, i_n], o))$ can be formulated by the following equation:

$$\begin{aligned} W_b.o &= W_b(i_1, \ldots, i_c, \ldots, i_n) \\ &= p(i_c)?W_a(i_1, \ldots, i_c, \ldots, i_n) : Fail \end{aligned} \tag{3.13}$$

Here the p?A:B notation means exactly *if p then return A else return B*.

For example, given a pair of numbers, W_5 shown in Figure 8(a) outputs the second number if it is greater than the first number; otherwise fail. Similarly, W_6 shown in Figure 8(b) outputs the second number if it is not greater than the first number; otherwise fail. Both W_5 and W_6 are created from the *Projection* workflow by applying the Conditional construct with input port i_1 designated as the *conditional port*. Because of the predicates on W_5 and W_6 are opposite, given the same inputs, only one of the workflows can be executed and the other one

will fail. For instance, given an input pair $[2, 3]$ for the conditional port and a number 2 which is used to specify the number in the pair to be projected, the outputs of W_5 and W_6 are:

$$\begin{aligned} W_5.o &= W_5([2,3],2) \\ &= (2<3)?Projection([2,3],2) : Fail \\ &= Projection([2,3],2) = 3 \\ W_6.o &= W_6([2,3],2) \\ &= (2>=3)?Projection([2,3],2) : Fail = Fail \end{aligned} \tag{3.14}$$

Traditional *if-then-else* statement and multiple-branch conditional statement can be supported by applying the Conditional construct on different branches of workflows.

3.3.5 The Loop Construct

The Loop construct enables cyclic executions of a workflow based on a predicate over the output of the workflow. The output of the workflow will be repetitively returned (fed back) to a specified input port until the predicate evaluates to true.

As illustrated in Figure 3.4(e), to apply the Loop construct on a workflow $W_a([i_1, \ldots, i_n], o)$, one of the input ports of W_a, $i_l \in [i_1, \ldots, i_n]$, is designated as the *loop port*, which takes input either from the initial input data product at the first iteration or the feedback from the output of the previous iteration. A user-input predicate p is set on the output port o such that if p evaluates to false, the output will be fed back to the *loop port*. Therefore, it is required that $dom(W_a.o) \subseteq dom(W_a.i_l)$. The semantics of the Loop construct

$L_{i_l,p(o)}(W_a([i_1,\ldots,i_n],o))$ can be formulated by the following recursive equation:

$$
\begin{aligned}
W_b.o &= W_b(i_1,\ldots,i_l,\ldots,i_n) \\
&= o_m \\
where\ \ o_1 &= W_a(i_1,\ldots,i_l,\ldots,i_n)\ \ (p(o_1)=false) \\
o_2 &= W_a(i_1,\ldots,o_1,\ldots,i_n)\ \ (p(o_2)=false) \\
&\ldots \\
o_m &= W_a(i_1,\ldots,o_{m-1},\ldots,i_n) \\
&\qquad (p(o_m)=true)
\end{aligned}
\tag{3.15}
$$

As an example, the workflow W_7 shown in Figure 3.9 repeatedly increase the base value by 1 until it is greater than 100. W_7 is created from a predefined workflow *Add* by applying the Loop construct with input ports i_1 designated as the loop port. Given input list $[0,1]$, the

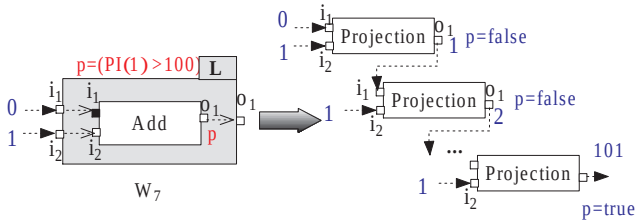


Figure 3.9: Workflow W_7 created by applying the Loop construct on an *Add* Workflow.

output of W_7 is:

$$
\begin{aligned}
o_1 &= Add(0,1) = 1 \;\; (o_1 \not> 100) \\
o_2 &= Add(1,1) = 2 \;\; (o_2 \not> 100) \\
&\cdots \\
o_{101} &= Add(100,1) = 101 \;\; (o_{101} > 100) \\
and \;\; W_7.o &= W_7(0,1) \\
&= o_{101} = 101
\end{aligned}
$$

.

3.3.6 The Curry Construct

The Curry construct allows users to fix one of the input ports with a specified argument and thus reduce the number of input ports. By applying multiple Curry constructs, a workflow that takes multiple arguments can be translated into a chain of workflows each with a single argument.

As illustrated in Figure 3.4(f), to apply the Curry construct on a workflow $W_a([i_1, \ldots, i_n], o)$, one of the input ports of W_a, $i_u \in [i_1, \ldots, i_n]$ is assigned with an argument. Therefore, the resulted workflow W_b will only have $n-1$ input ports and there are one to one mappings from those ports to the input ports of W_a.

The semantics of the Curry construct $U_{i_u,p}(W_a([i_1, \ldots, i_u, \ldots, i_n], o))$ can be formulated by the following equation:

$$
\begin{aligned}
W_b.o &= W_b(d_1, \ldots, d_{n-1}) \\
&= W_a(d_1, \ldots, p, \ldots, d_{n-1})
\end{aligned}
\tag{3.16}
$$

As an example, the workflow W_8 shown in Figure 3.10 implements the Increment operator by applying The Curry construct on the Addition workflow. W_8 contains only one input port and will automatically increment the input integer by 1.

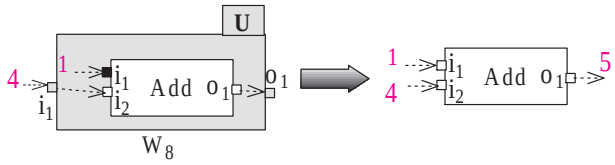


Figure 3.10: Workflow W_8 created by applying the Curry construct on an *Add* Workflow.

Theorem 3.3.4 *The Curry construct satisfies* commutativity*:*

$$\begin{aligned} & U_{i_{u_1},p_1}(U_{i_{u_2},p_2}(W_a([i_1,\ldots,i_n],o))) \\ =& U_{i_{u_2},p_2}(U_{i_{u_1},p_1}(W_a([i_1,\ldots,i_n],o))) \\ & (i_{u_1},i_{u_2}\in[i_1,\ldots,i_n],i_{u_1}\neq i_{u_2}) \end{aligned} \tag{3.17}$$

Proof: Let $W_b = U_{i_{u_2},p_2}(W_a)$ and $W_c = U_{i_{u_1},p_1}(W_b)$

Then according to equation 3.16, for any given inputs $i_1,\ldots,i_n$,we have

$$\begin{aligned} W_c.o &= W_b(i_1,\ldots,p_2,\ldots,i_n) \\ &= W_a(i_1,\ldots,p_1,\ldots,p_2,\ldots,i_n) \end{aligned} \tag{3.18}$$

Similarly, let $W_d = U_{i_{u_1},p_1}(W_a)$ and $W_e = U_{i_{u_2},p_2}(W_d)$, we can get

$$\begin{aligned} W_e.o &= W_d(i_1,\ldots,p_1,\ldots,i_n) \\ &= W_a(i_1,\ldots,p_1,\ldots,p_2,\ldots,i_n) \end{aligned} \tag{3.19}$$

From equation 3.18 and 3.19, for any given inputs $i_1,\ldots,i_n$, we can conclude equation 3.17.

□

Theorem 3.3.5 *The Curry construct is* commutative *with all unary constructs:*

$$\begin{aligned} & U_{i_u,p}(M_{i_m}(W_a([i_1,\ldots,i_n],o))) \\ =& M_{i_m}(U_{i_u,p}(W_a([i_1,\ldots,i_n],o))) \\ & (i_u, i_m \in [i_1,\ldots,i_n], i_u \neq i_m) \end{aligned} \tag{3.20}$$

$$\begin{aligned} & U_{i_u,p}(R_{i_b,i_r}(W_a([i_1,\ldots,i_n],o))) \\ =& R_{i_b,i_r}(U_{i_u,p}(W_a([i_1,\ldots,i_n],o))) \\ & (i_u, i_b, i_r \in [i_1,\ldots,i_n], i_u \neq i_b, i_u \neq i_r) \end{aligned} \tag{3.21}$$

$$\begin{aligned} & U_{i_u,p}(T_{i_l,i_r}(W_a([i_1,\ldots,i_n],o))) \\ =& T_{i_l,i_r}(U_{i_u,p}(W_a([i_1,\ldots,i_n],o))) \\ & (i_u, i_l, i_r \in [i_1,\ldots,i_n], i_u \neq i_l, i_u \neq i_r) \end{aligned} \tag{3.22}$$

$$\begin{aligned} & U_{i_u,p}(C_{p(i_c)}(W_a([i_1,\ldots,i_n],o))) \\ =& C_{p(i_c)}(U_{i_u,p}(W_a([i_1,\ldots,i_n],o))) \\ & (i_u, i_c \in [i_1,\ldots,i_n], i_u \neq i_c) \end{aligned} \tag{3.23}$$

$$\begin{aligned} & U_{i_u,p}(L_{i_l,p(o)}(W_a([i_1,\ldots,i_n],o))) \\ =& L_{i_l,p(o)}(U_{i_u,p}(W_a([i_1,\ldots,i_n],o))) \\ & (i_u, i_l \in [i_1,\ldots,i_n], i_u \neq i_l) \end{aligned} \tag{3.24}$$

Due to the page limit, we will skip those proofs which will be similar to the one for Theorem 3.3.4. Theorem 3.3.4 and Theorem 3.3.5 are important foundations that allow users to set parameters to arbitrary Curry based workflows and the parameters can be correctly passed to the enclosed base workflow. Below we will prove that Corollary 3.3.3 can be extended to workflows with multiple input ports.

Theorem 3.3.6 *Given a workflow $W_a([i_l, \ldots, i_l, \ldots, i_r, \ldots, i_n], o)$ that is associative with port i_l and i_r, the unary-construct-based workflow $W_b = T_{i_l,i_r}(W_a)$ satisfies the following equation for any inputs $p_1, \ldots, [d_1, \ldots, d_m], \ldots, p_n$:*

$$\begin{aligned} & W_b(p_1, \ldots, [d_1, \ldots, d_m], \ldots, p_n) \\ =& W_a(p_1, \ldots \\ & W_a(p_1, \ldots, W_a(p_1, \ldots, d_1, \ldots, d_2, \ldots, p_n), \ldots, d_3, \ldots, p_n) \\ & \qquad \ldots, d_n, \ldots, p_n) \end{aligned} \tag{3.25}$$

Proof: We first apply multiple Curry constructs on W_a and each will assign an parameter to one of the input ports except i_l and i_r. W_a can therefore be translated to a chain of workflows as following:

$$\begin{aligned} W_{a_1} &= U_{i_1,p_1}(W_a) \\ W_{a_2} &= U_{i_2,p_2}(W_{a_1}) \\ &\ldots \\ W_{a_{n-2}} &= U_{i_n,p_n}(W_{a_{n-3}}) = U_{i_n,p_n}(\ldots U_{i_1,p_1}(W_a)\ldots) \end{aligned} \tag{3.26}$$

By doing this, we can obtain a binary workflow $W_{a_{n-2}}([i_l, i_r], o)$ that is associative. Then according to Corollary 3.3.3, the Tree based workflow $W_{b'} = T_{i_l,i_r}(W_{a_{n-2}})$ satisfies the following equation for any input list $[d_1, \ldots, d_n]$:

$$\begin{aligned} & W_{b'}([d_1, \ldots, d_m]) = \\ & \qquad W_{a_{n-2}}(\ldots W_{a_{n-2}}(W_{a_{n-2}}(d_1, d_2), d_3)\ldots, d_m) \end{aligned} \tag{3.27}$$

According to equation 3.22, we can derive

$$\begin{aligned} W_{b'} &= T_{i_l,i_r}(W_{a_{n-2}}) \\ &= U_{i_n,p_n}(T_{i_l,i_r}(W_{a_{n-3}})) \\ &\dots \\ &= U_{i_n,p_n}(\dots U_{i_1,p_1}(T_{i_l,i_r}(W_a))\dots) \\ &= U_{i_n,p_n}(\dots U_{i_1,p_1}(W_b)\dots) \end{aligned} \tag{3.28}$$

Therefore by equation 3.28 and 3.16 we can get

$$W_b(p_1,\dots,[d_1,\dots,d_m],\dots,p_n) = W_{b'}([d_1,\dots,d_m]) \tag{3.29}$$

From equation 3.27 and 3.29 we can get

$$\begin{aligned} &W_b(p_1,\dots,[d_1,\dots,d_m],\dots,p_n) = \\ &\qquad W_{a_{n-2}}(\dots W_{a_{n-2}}(W_{a_{n-2}}(d_1,d_2),d_3)\dots,d_m) \end{aligned} \tag{3.30}$$

From equation 3.26 and 3.16, we can derive

$$\begin{aligned} &W_{a_{n-2}}(\dots W_{a_{n-2}}(W_{a_{n-2}}(d_1,d_2),d_3)\dots,d_m) \\ =&W_a(p_1,\dots \\ &W_a(p_1,\dots,W_a(p_1,\dots,d_1,\dots,d_2,\dots,p_n),\dots,d_3,\dots,p_n) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \dots,d_n,\dots,p_n) \end{aligned} \tag{3.31}$$

From equation 3.30 and 3.31 we can conclude equation 3.25.

□

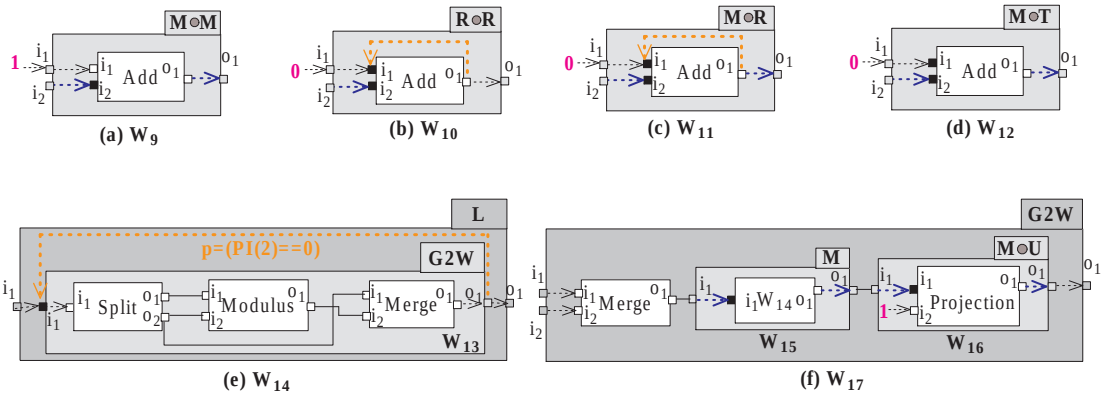


Figure 3.11: (a) unary-construct-based workflow W_9 created by the composition of two Map constructs on the *Add* workflow; (b) Unary-construct-based workflow W_{10} created by the composition of two Reduce constructs on the *Add* workflow; (c) unary-construct-based workflow W_{11} created by the composition of the Map construct and the Reduce construct on the *Add* workflow; (d) unary-construct-based workflow W_{12} created by applying the composition of the Map construct and the Tree construct on the *Add* workflow; (e) unary-construct-based workflow W_{15} created by applying the Loop construct on a graph-based workflow; and, (f) graph-based workflow W_{17} created by applying the $G2W$ construct on a workflow graph.

3.4 Workflow Composition

Comparing with existing workflow composition models, our model has the following novel characteristics:

1. Workflows are the only operands for workflow composition. All composite workflows are created as the composition of existing workflows.

2. Every workflow can be directly used for workflow composition through workflow constructs and every composition results in a new workflow, either a graph-based workflow or a unary-construct-based workflow.

3. Workflow constructs are fully composable and the set of workflows is closed under all the workflow constructs.

These characteristics makes our framework unique in the ability to apply the proposed workflow constructs and their compositions on arbitrary workflows, as illustrated by the following scenarios.

Unary workflow constructs can compose with each other arbitrarily. Given an existing unary-construct-based workflow $W_b = S_1(W_a)$ which is created by applying a unary construct S_1 on a workflow W_a, we can apply another unary construct S_2 on W_b resulting in $W_c = S_2(W_b) = S_2(S_1(W_a))$. We define a composition of S_1 and S_2 as a new unary construct to simplify this two-step composition. The semantics of this new unary construct is given by the following formula:

$$(S_2 \circ S_1)(W_a) = S_2(S_1(W_a)) \tag{3.32}$$

For example, given a predefined workflow *Add*, applying different compositions of Map and Reduce constructs will result in different workflows. W_9 shown in Figure 3.11(a) is created by applying the composition of two Map constructs. Given inputs of a base value 1 and a table of numbers (represented as a list of list), W_9 will increase all the numbers in the table by one and output the resulting table. W_{10} shown in Figure 3.11(b) is created by applying the composition of two Reduce constructs. Given inputs of a base value 0 and a table of numbers, W_{10} will output a sum of the whole table. W_{11} shown in Figure 3.11(c) is created by applying the composition of Map and Reduce. Given inputs of a base value 0 and a $m \times n$ table of numbers, W_{11} will output a list of m numbers, each representing the sum of the corresponding row in the table. The composition of unary constructs are arbitrary. Any finite number of application of Map and Reduce constructs is allowed, which enables the processing of data cubes in any dimensions. W_{12} shown in Figure 3.11(d) is created by applying the composition of Map and Tree. Given the same inputs, W_{11} and W_{12} will have the same output. However, W_{12} supports parallel aggregation using the Tree construct.

Unary workflow constructs can compose with other constructs arbitrarily and hierarchically. In particular, our unary workflow constructs can be applied to a graph-based workflow to form a unary-construct-based workflow; several unary-construct-based workflows can also be linked together by data channels to form a workflow graph G and then the $G2W$ construct can be applied to G to form a graph-based workflow.

As an example, W_{14} shown in Figure 3.11(e) implements the Euclidean algorithm to calculate the greatest common divisor for a pair of integers. W_14 is created by applying the Loop construct on a graph-based workflow W_13. Given a pair of integers $[a, b]$ as input, W_8 will output a pair of integers $[b, a\%b]$. By the application of the Loop construct, W_13 will be executed repeatedly until the predicate $p = (PI(2) == 0)$ evaluates to be true which means the second number of the pair equals to 0. Finally, W_14 will output a pair of integers $[gcd(a, b), 0]$, where $gcd(a, b)$ is the greatest common divisor of the input pair. A unary-based workflow W_15 can then be created by applying the Map construct on W_14. W_15 takes a list of pairs $[[a_1, b_1], ..., [a_n, ...b_n]]$ and outputs a list of pairs $[[gcd(a_1, b_1), 0], ..., [gcd(a_n, b_n), 0]]$.

Further, W_{17} shown in Figure 3.11(f) can process two lists in parallel and calculate the greatest common divisor for each corresponding pair in two lists. Given two lists $[a_1, ..., a_n]$ and $[b_1, ..., b_n]$ as input, W_{17} will output a list $[gcd(a_1, b_1), ..., gcd(a_n, b_n)]$ containing the greatest common divisors for each corresponding pair. W_{17} is created by applying the $G2W$ construct on a workflow graph which contains three workflows. The two input lists are merged into one list of pairs $[[a_1, b_1], ..., [a_n, ...b_n]]$ by the *Merge* workflow and sent to W_{15}. W_{15} then outputs a list of pairs $[[gcd(a_1, b_1), 0], ..., [gcd(a_n, b_n), 0]]$ to W_{16} which is created by applying the Map construct and Curry construct on a $Projection$ workflow. W_{16} will project the first element in each pair resulting in a list $[gcd(a_1, b_1), ..., gcd(a_n, b_n)]$.

3.5 A Dataflow Based Approach for Exception Handling

Much research has been carried out on issues of exception handling in workflow management systems [18], [106], [36], [108], [12]. However, most of the existing approaches are rule or

event based. In this section, we propose a dataflow based approach for exception handling which is compatible with our scientific workflow composition model.

3.5.1 Exception Handling

In our approach, an exception is represented as a special data product (called exception data product) which contains exception information. As shown in Figure 3.12, each workflow contains a default exception port as the output port specifically for exception data product. exception port can be linked to exception handling workflows such as Stop, Pause or user defined handlers.

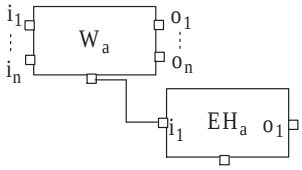


Figure 3.12: Workflow exception handling.

As the basic building block, a primitive workflow is responsible to capture all the exceptions during the invocation of inside tasks, generate corresponding workflow exception data products and output through the exception port. Workflow exceptions can also be propagated hierarchically to higher level composite workflows following the workflow construction. As shown in Figure 3.13, the exception ports of W_a or W_b are automatically mapped to the exception port of W_c. Therefore, whenever an exception data product e_1 is generated by either W_a or W_b, it will be passed to W_c, and W_c will generate a new exception data product e_2 which contains e_1 as well as the information of W_c.

3.5.2 The Exception Construct

The data exception construct enables the user to capture the data exception on one of the input/output ports. As illustrated in Figure 3.14, to apply the Exception construct to a workflow

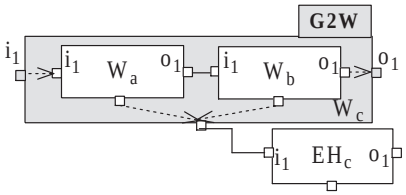


Figure 3.13: Workflow exception propagation.

$W_a([i_1, \ldots, i_n], o)$, one of the input/output ports is designated as the *exception test port*, on which a logical test will be calculated based on the input/output data product. A user-input predicate p is set on the *exception test port* and a user-defined exception data product e needs to be designated to the exception port. If p evaluates to be true, W_b will behave exactly as W_a, otherwise W_b will output e from the exception port.

As an example shown in Figure 3.15, W_{18} detects the typical division by zero error and outputs an exception data product.

3.6 Case Studies

The proposed techniques have been realized in an XML-based scientific workflow specification language, called WSL [53], and implemented in a new version of the VIEW system [85]. The implementation details will be discussed in Chapter 5. However, we will present several case studies in order to validate our techniques.

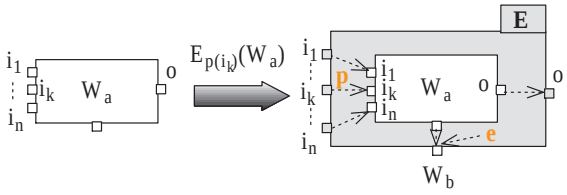


Figure 3.14: The exception construct.

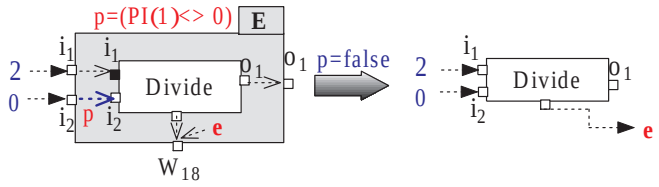


Figure 3.15: Workflow W_{18} created by applying the Exception construct on a Divide workflow.

3.6.1 Workflow for Freebase Processing

We implemented a Freebase Processing Workflow as shown in Figure 3.16 to validate the ability of our model to leverage MapReduce tasks to the workflow level. We choose Amazon Elastic MapReduce [1] for this case study. Amazon Elastic MapReduce is a Web service that utilizes a hosted Hadoop framework running on the web-scale infrastructure. Amazon Elastic MapReduce published three sample job flows [6] which are used to filter a set of Freebase data and store it into Amazon SimpleDB data store. In our experiment, we created three primitive workflows: WFreequentID, WDataStorage, and WNameStorage, and each is based on one of the job flows. The WFreequentID workflow can iterate over each file of input to look for the most popular Freebase IDs. The WDataStorage workflow stores the results of the WFreequentID workflow into Amazon SimpleDB. The WNameStorage workflow reads Freebase data and stores names and their IDs into Amazon SimpleDB. We then created a graph-based Freebase Processing Workflow which is composed by those three workflows. Our workflow technique automatically connects the execution of the three workflows via

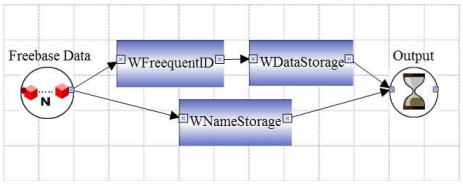


Figure 3.16: Freebase Processing Workflow.

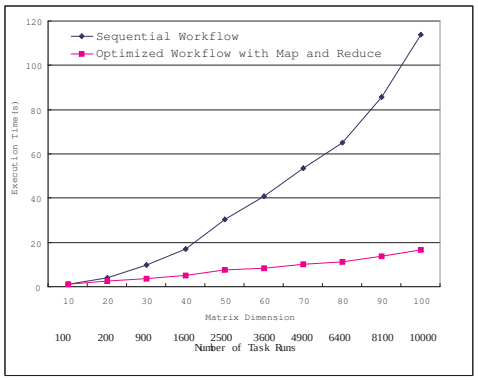


Figure 3.17: Performance comparison of two workflows for matrix summation.

explicit dataflows, and naturally enables concurrent execution of the WFreequentID workflow and WNameStorage workflow as they do not have data dependencies.

3.6.2 Workflows for Matrix Summation

We designed two workflows for matrix summation in order to validate the performance of our Map construct. The first workflow sequentially adds up all the elements in the matrix. The second workflow takes advantage of the Map construct and calculates the summations for each row in parallel, and then sums up the results using a Reduce-based addition workflow. We add a delay of 10 ms for each addition task for better observation and run the two workflows on 10 matrixes with different sizes. Figure 3.17 compares the performance of the two workflows. The efficiency of the first workflow is $O(n^2)$ while the efficiency of the second workflow is $O(n)$ (n represents the dimension of the matrix).

3.7 Chapter Summary

In this chapter, a dataflow-based scientific workflow composition model was proposed. Comparing with existing workflow composition models, our approach clearly separates tasks from the workflow composition layer and thus elegantly solves the two-world problem in existing

composition frameworks. Based on such a novel model, our proposed operators are fully composible one with another and can be applied on arbitrary workflows.

CHAPTER 4

COLLECTIONAL DATA MODEL

Modeling, organizing, and processing scientific data have become key challenges for scientific workflow management systems (SWFMSs). While business data are usually relational, scientific data are typically hierarchical and collection-oriented. a motivating example is introduced in Section 4.1 to illustrate this problem. In this chapter, we take a first step toward formalizing a collectional data model for scientific data processing, which is defined in Section 4.2. While the relational data model is based on the notion of relation, we introduce the term *collectional* in our proposed collectional data model to emphasize that our data model is built on the notion of collection. Section 4.3 briefly discusses the application of the collectional model in scientific workflow compositions. Finally, Section 4.4 summarize this chapter.

4.1 An Motivating Example of Biological Simulation

The marine worm *Nereis succinea* spawns during a coordinated "nuptial dance" timed by the phases of the moon, initiated by the time of the day, and choreographed by the exchange of chemical signals [101]. Females excrete a pheromone that can be attractants for the opposite sex in many environments. We developed a biological simulation project, called TangoInSilico [52], for testing the hypothesis that male responses to low concentrations of CSSG can facilitate finding females.

The simulation model consists of more than twenty parameters, e.g., concentration of pheromone, initial degree of the male worm. Scientists need to run the same *Simulation* workflow thousands of times with different combinations of parameter sets in order to adjust the parameters and test the hypothesis. One key challenge of this project is to systematically

manage the large set of parameters and results in order to facilitate scientists to do statistic analysis and scientific queries. We adopted a list oriented approach in the previous scientific workflow composition model [53]. Although the list structure supports the parallel processing with our workflow constructs, it cannot effectively organize the hierarchical and table-like parameter datasets, and limited querying and data manipulation power. In this chapter, we propose a collectional data model and apply it to this project to organize the input parameter data sets and output results, which will be shown in examples later in this chapter.

4.2 The Collectional Data Model

Following the terminology in the relational model [39], a datum is associated with a *domain*. For the purpose of this dissertation, we restrict the set of atomic domains: $dom(D) =$ *String* | *Integer* | *Double* | *Boolean* | *File*. Here we use the notation $dom(D)$ to denote the domain name of datum D. More atomic domains can be easily introduced for different applications.

We briefly review the relational model. A **relation** $\mathfrak{R}$ is a pair $< R, r >$ where R is a schema of the relation and r is an instance of that schema. a **relation instance** is a table with rows (called *tuples*) and named columns (called *attributes*). A **relation schema** can be defined as an unordered tuple $< c_1 : d_1, c_2 : d_2, ..., c_n : d_n >$ where $c_1, c_2, ..., c_n$ are column names and $d_1, d_2, ..., d_n$ are domain names. The values that appear in the corresponding columns must belong to the specified domains. As a special case, *a constant data value can be defined as a single-column-single-row relation*.

Based on the relational model, we propose a collectional model with relations as building blocks. First we introduce the central construct in our model, the *collection*.

Definition 4.2.1 (Collection) A **collection** $\mathfrak{C}$ is a tuple $< C, c >$ where C is a *collection schema* and c is a *collection instance* of that schema. $\diamondsuit$

Definition 4.2.2 (Collection Schema) A **collection schema** is a pair $< K, V >$ where

- K, the **key**, is a pair $k : d$ where k is the *key name* and d is the *domain name*.

- V, the **value**, is either a relation schema or a collection schema.

$\diamondsuit$

A collection is nested if V is a collection schema. Intuitively, a nested collection can be considered as a tree-like structure in which all the leaves are relations and each *level* of the collection is identified by a unique key.

Definition 4.2.3 (Height) The **height** of a collection schema $C = < K, V >$, denoted by $\mathcal{H}(C)$, is defined as follows:

- If V is a relation schema, then $\mathcal{H}(C) = 1$.

- Otherwise, $\mathcal{H}(C) = \mathcal{H}(V) + 1$.

$\diamondsuit$

More generally, we use an expanded notation $< k_1 : d_1, k_2 : d_2, ..., k_n : d_n, R >$ to represent a collection schema where R is a relation schema. Without loss of generality, we restrict that *different keys within one collection must have distinct key names although they can share the same domain*. We define $\mathcal{K}_i(C)$ $(i \in \{1, ..., n\})$ as a function that returns k_i.

Definition 4.2.4 (Collection Instance) A **collection instance** is a set of key-value pairs $(p_i, q_i)(i \in \{1, ..., m\})$ where m is the number of pairs in the set, called the **cardinality** of the collection instance, each q_i is either a relation instance or a collection instance and $dom(p_i) =$ *String* | *Integer* | *Double* | *Boolean*. $\diamondsuit$

Definition 4.2.5 *A collection instance c* ***conforms to*** *the collection schema $C < k : d, V >$, denoted by $c \models C$ iff:*

$\forall (p_i, q_i)(i \in \{1, ..., m\}) \in c, dom(p_i) = d, p_i \models V.$

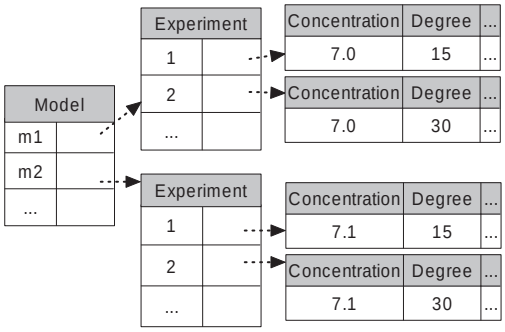


Figure 4.1: The *Parameters* collection.

A collection $\mathfrak{C}$ is **valid** iff the collection instance *conforms to* the collection schema.

As an example, Figure 4.1 illustrates an instance of the collection *Parameters* with the collection schema $< Model$: ***String***$, Experiment$: ***Integer***$, < Concentration$: ***Double***$, Degree$: ***Integer*** $>>$. The schema states that the height of the *Parameters* collection is 2: the first level contains a key named *Model* and belongs to domain *String*; the second level contains a key named *Experiment* and belongs to domain *Integer*; and all the inner relation instances satisfy the relation schema $< Concentration$: ***Double***$, Degree$: ***Integer*** $>$.

The new collectional model naturally extends the relational model. We then extend the relational operators to collectional operators of which *collections are the only operands*. Note that *a relation can be defined as a collection whose height and cardinality are equal to 1, and all collectional operators will then be reduced to their relational counterparts*. Below, we first extend *union* and *set difference*, and then extend *selection*, *projection*, *Cartesian product* and *renaming*.

Since collections are sets, the set operators are applicable to collections. However, similarly to relational algebra, the union and the set difference operators cannot be applied on arbitrary collections. We therefore limit the scope of the union and the set difference operators and apply them only on *union-compatible* collections, which are defined as follows:

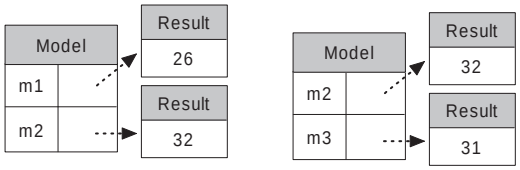


Figure 4.2: Two collections that union-compatible : (a) collection M_1 ; and (b) collection M_2 .

Definition 4.2.6 *Two collection schemas* $C_1 =< k_1 : d_1, S_1 >$ *and* $C_2 =< k_2 : d_2, S_2 >$ *are equal, denoted* $C_1 = C_2$ *iff* $k_1 = k_2$, $d_1 = d_2$, *and* $S_1 = S_2$.

Definition 4.2.7 *Two collections* $\mathfrak{C}_1 =< C_1, c_1 >$ *and* $\mathfrak{C}_2 =< C_2, c_2 >$ *are* union-compatible *iff* $C_1 = C_2$.

As an example, Figure 4.2 illustrates two collections M_1 and M_2 that are union-compatible.

Union ($\cup^c$) Union is a binary operator that calculates the union of two collections. More specifically, given two collections $\mathfrak{C}_1 =< C_1, c_1 >$ and $\mathfrak{C}_2 =< C_2, c_2 >$ as inputs, a union operation is specified as $\mathfrak{C}_1 \cup^c \mathfrak{C}_2$, resulting in a collection $\mathfrak{C}' =< C', c' >$ where the schema

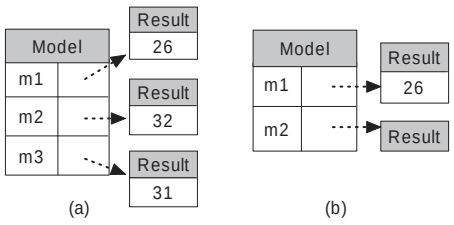


Figure 4.3: The results of (a) $M_1 \cup^c M_2$; and (b) $M_1 -^c M_2$.

$C' = C_1 = C_2$, and:

$$\begin{aligned} c' = & \{(p, q_1 \cup^c q_2) \mid (p, q_1) \in c_1 \wedge (p, q_2) \in c_2\} \\ & \cup \{(p, q_1) \mid (p, q_1) \in c_1 \wedge \neg\exists q_2\ (p, q_2) \in c_2\} \\ & \cup \{(p, q_2) \mid (p, q_2) \in c_2 \wedge \neg\exists q_1\ (p, q_1) \in c_1\}\ if(\mathcal{H}(C_1) > 1) \end{aligned} \tag{4.1}$$

$$\begin{aligned} & \{(p, q_1 \cup q_2) \mid (p, q_1) \in c_1 \wedge (p, q_2) \in c_2\} \\ & \cup \{(p, q_1) \mid (p, q_1) \in c_1 \wedge \neg\exists q_2\ (p, q_2) \in c_2\} \\ & \cup \{(p, q_2) \mid (p, q_2) \in c_2 \wedge \neg\exists q_1\ (p, q_1) \in c_1\}\ if(\mathcal{H}(C_1) = 1) \end{aligned}$$

The union operator satisfies both commutativity and associativity:

$$\mathfrak{C}_1 \cup^c \mathfrak{C}_2 = \mathfrak{C}_2 \cup^c \mathfrak{C}_1 \tag{4.2}$$

$$(\mathfrak{C}_1 \cup^c \mathfrak{C}_2) \cup^c \mathfrak{C}_3 = \mathfrak{C}_1 \cup^c (\mathfrak{C}_2 \cup^c \mathfrak{C}_3) \tag{4.3}$$

Figure 4.3(a) illustrates the union of M_1 and M_2 which contains all the results of both collections with duplications eliminated.

Set difference ($-^c$) Set difference is a binary operator that calculates the difference of two collections. More specifically, given two collections $\mathfrak{C}_1 =< C_1, c_1 >$ and $\mathfrak{C}_2 =< C_2, c_2 >$ as inputs, a set difference operation is specified as $\mathfrak{C}_1 -^c \mathfrak{C}_2$ resulting in a collection $\mathfrak{C}' =< C', c' >$ where the schema $C' = C_1 = C_2$, and:

$$\begin{aligned} c' = & \{(p, q_1) \mid (p, q_1) \in c_1 \wedge \neg\exists q_2\ (p, q_2) \in c_2\} \\ & \cup \{(p, q_1 -^c q_2) \mid (p, q_1) \in c_1 \wedge (p, q_2) \in c_2\} if(\mathcal{H}(C_1) > 1) \end{aligned} \tag{4.4}$$

$$\begin{aligned} & \{(p, q_1) \mid (p, q_1) \in c_1 \wedge \neg\exists q_2\ (p, q_2) \in c_2\} \\ & \cup \{(p, q_1 - q_2) \mid (p, q_1) \in c_1 \wedge (p, q_2) \in c_2\}\ if(\mathcal{H}(C_1) = 1) \end{aligned}$$

Figure 4.3(b) illustrates the difference of M_1 and M_2 which returns the items that belong to M1 but not to M2.

Selection (σ^c) Selection is a unary operator that selects the elements which satisfy the selection condition. More specifically, given a collection $\mathfrak{C} =< C, c >$ as input where the schema $C =< k_1 : d_1, k_2 : d_2, ..., k_n : d_n, R >$, a selection operation is specified as $\sigma^c_\varphi(\mathfrak{C})$ where φ is the selection condition represented as a propositional formula that consists of atoms and logical operators $\wedge$ (and), $\vee$ (or), and $\neg$ (negation).

An atom can be any one of the following:

- $k\theta v$ where $k \in k_1, ...k_n$ is a key name and v is a constant value belonging to the same domain of key k.
- $a\theta b$ where a and b are attribute names of R.
- $a\theta v$ where a is an attribute name of R and v is a constant value belonging to the same domain of attribute a.

where θ is a binary operator in the set $<, \leq, =, >, \geq$.

The schema of the resultant collection is equivalent to C, and we define the selection operator at the instance level as follows:

$$\sigma^c_{k\theta v}(c) = \begin{cases} \{(p,q) \mid (p,q) \in c \wedge p\theta v\} & if(\mathcal{K}_1(C) = k) \\ \{(p, \sigma^c_{k\theta v}(q)) \mid (p,q) \in c\} & if(\mathcal{K}_1(C) \neq k) \end{cases} \tag{4.5}$$

$$\sigma^c_{a\theta b}(c) = \begin{cases} \{(p, \sigma^c_{a\theta b}(q)) \mid (p,q) \in c\} & if(\mathcal{H}(C) > 1) \\ \{(p, \sigma_{a\theta b}(q)) \mid ((p,q) \in c)\} & if(\mathcal{H}(C) = 1) \end{cases} \tag{4.6}$$

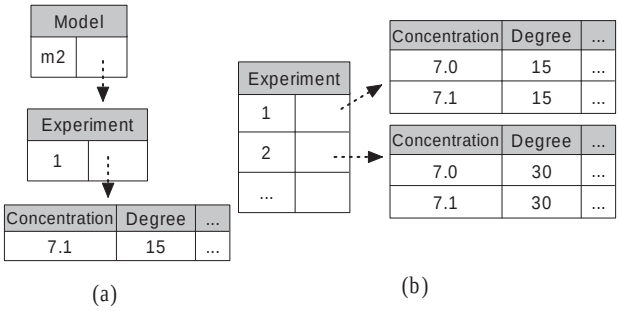


Figure 4.4: The results of the selection and projection operations (a) $\sigma^c_{Model='m2'ANDExperiment='1'}\ (Parameters)$; and (b) $\pi^c_{Experiment}\ (Parameters)$.

$$\sigma^c_{a\theta v}(c) = \begin{cases} \{(p, \sigma^c_{a\theta v}(q)) \mid (p,q) \in c\} & if(\mathcal{H}(C) > 1) \\ \{(p, \sigma_{a\theta v}(q)) \mid ((p,q) \in c)\} & if(\mathcal{H}(C) = 1) \end{cases} \tag{4.7}$$

The selection operator satisfies the following properties:

$$\sigma^c_A(\mathfrak{C}) = \sigma^c_A \sigma^c_A(\mathfrak{C}) \tag{4.8}$$

$$\sigma^c_A \sigma^c_B(\mathfrak{C}) = \sigma^c_B \sigma^c_A(\mathfrak{C}) \tag{4.9}$$

$$\sigma^c_{A \wedge B}(\mathfrak{C}) = \sigma^c_A(\sigma^c_B(\mathfrak{C})) = \sigma^c_B(\sigma^c_A(\mathfrak{C})) \tag{4.10}$$

$$\sigma^c_{A \vee B}(\mathfrak{C}) = \sigma^c_A(\mathfrak{C}) \cup \sigma^c_B(\mathfrak{C}) \tag{4.11}$$

Figure 4.4(a) illustrates a selection of the $Parameters$ collection which selects the parameter set of the experiment '1' under model 'M1'.

Projection (π^c) Projection is a unary operator that extracts sub-collections from the input collection. More specifically, given a collection $\mathfrak{C} =< C, c >$ as input where the schema C is $< k_1 : d_1, k_2 : d_2, ..., k_n : d_n, R >$, a projection operation is specified as $\pi^c_\psi(\mathfrak{C})$ in which ψ is a sequence in the form of $k, a_1, ...a_n$ where:

- $k \in k_1, ...k_n$ is a key name.

- $a_1, ...a_r$ are a sequence (zero or more) of attribute names in the relation schema R.

The schema of the resultant collection is $< k_i : d_i, k_{i+1} : d_{i+1}, ..., k_n : d_n, \pi_{a_1,...,a_r}(R) >$ where $k_i = k$ and $\pi_{a_1,...,a_r}(R')$ removes all the attributes except $a_1, ...a_r$ as defined in traditional relational algebra. By the definition, we can conclude:

Theorem 4.2.8 *Given two collection schemas C_1 and C_2, if $C_1 = C_2$, then $\pi^c_\psi(C_1) = \pi^c_\psi(C_2)$*

We then define the projection operator at the instance level as follows:

$$\pi^c_k(c) = \begin{array}{ll} c & if(\mathcal{K}_1(C) = k) \\ \cup^c_{(p,q)\in c}\, \pi^c_k(q) & if(\mathcal{K}_1(C) \neq k) \end{array} \tag{4.12}$$

$$\pi^c_{a_1,...,a_r}(c) = \begin{array}{ll} \{(p, \pi^c_{a_1,...,a_r}(q)) \mid (p,q) \in c\} & if(\mathcal{H}(C) > 1) \\ \{(p, \pi_{a_1,...,a_r}(q)) \mid (p,q) \in c\} & if(\mathcal{H}(C) = 1) \end{array} \tag{4.13}$$

$$\pi^c_{k,a_1,...,a_r}(c) = \pi^c_k(\pi^c_{a_1,...,a_r}(c)) = \pi^c_{a_1,...,a_r}(\pi^c_k(c)) \tag{4.14}$$

The projection operator satisfies the following properties:

$$\pi^c_{k_2}(\pi^c_{k_1}(\mathfrak{C})) = \pi^c_{k_2}(\mathfrak{C}) \quad \text{where} \quad k_1 = \mathcal{K}_i(C), k_2 = \mathcal{K}_j(C), i <= j \tag{4.15}$$

$$\pi^c_{a_1,...,a_n}(\pi^c_{b_1,...,b_m}(\mathfrak{C})) = \pi^c_{a_1,...,a_n}(\mathfrak{C}) \quad \text{where} \quad \{a_1, ..., a_n\} \subseteq \{b_1, ..., b_m\} \tag{4.16}$$

Figure 4.4(b) illustrates a projection of the $Parameters$ collection in which the 'Model' key is deleted and the 'Experiment' key becomes the root level.

Cartesian product ($\times^c$) In order to define the Cartesian product between two collections, we first define the Cartesian product between a relation and a collection. Given a relation $\mathfrak{R} =< R, r >$ and a collection $\mathfrak{C} =< C, c >$ where $C =< k_1 : d_1, k_2 : d_2, ..., k_n : d_n, R_c >$, the operation $\mathfrak{R} \times^{rc} \mathfrak{C}$ returns a collection $\mathfrak{C}_r =< C_r, c_r >$ where $C_r =< k_1 : d_1, k_2 : d_2, ..., k_n : d_n, R \times R_c >$ and

$$c_r = \begin{cases} \{(p, r \times^{rc} q) \mid (p, q) \in c_r\} & if(\mathcal{H}(C_r) > 1) \\ \{(p, r \times q) \mid (p, q) \in c_r\} & if(\mathcal{H}(C_r) = 1) \end{cases} \tag{4.17}$$

Now we are able to define the Cartesian product between two collections. Given two collections $\mathfrak{C}_1 =< C_1, c_1 >$ and $\mathfrak{C}_2 =< C_2, c_2 >$ as inputs, a Cartesian product operation can be specified as $\mathfrak{C}_1 \times^c \mathfrak{C}_2$ where $C_1 =< k_1 : d_1, k_2 : d_2, ..., k_n : d_n, R_1 >$ and $C_1 =< l_1 : o_1, l_2 : o_2, ..., l_m : o_m, R_2 >$, resulting in a collection $\mathfrak{C}' =< C', c' >$. The resultant schema $C' =< k_1 : d_1, k_2 : d_2, ..., k_n : d_n, l_1 : o_1, l_2 : o_2, ..., l_m : o_m, R_1 \times R_2 >$ where $R_1 \times R_2$ contains all the attributes of both R_1 and R_1 as defined in the relational algebra. By the definition, we can conclude the following equation:

$$\mathcal{H}(C') = \mathcal{H}(C_1) + \mathcal{H}(C_2) \tag{4.18}$$

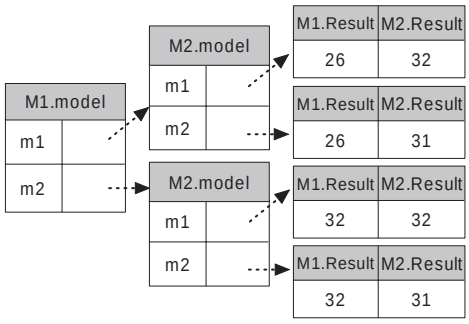


Figure 4.5: The result of the composition of the Cartesian product and the renaming operations $\rho^c_{M1.Model/Model}\ (\rho^c_{M1.Result/Result}\ (M_1)) \times^c \rho^c_{M2.Model/Model}\ (\rho^c_{M2.Result/Result}\ (M_2))$.

The resultant collection instance c' is defined by the following equation:

$$\begin{aligned} c' = &\{(p, q \times^c c_2) \mid (p, q) \in c_1\} \qquad if(\mathcal{H}(C_1) > 1) \\ &\{(p, q \times^{rc} c_2) \mid (p, q) \in c_1\} \qquad if(\mathcal{H}(C_1) = 1) \end{aligned} \tag{4.19}$$

Our collectional model is an ordered model and our Cartesian product operator does not satisfy commutativity. However, our Cartesian product satisfies associativity:

$$(\mathfrak{C}_1 \times^c \mathfrak{C}_2) \times^c \mathfrak{C}_3 = \mathfrak{C}_1 \times^c (\mathfrak{C}_2 \times^c \mathfrak{C}_3) \tag{4.20}$$

Similar to the relational model, naming conflicts can arise in some cases. For example, given two collections M_1 and M_2 in Figure 4.2 which contain the same key names, the resultant collection of operation $M_1 \times^c M_2$ will have a naming problem as it contains duplicate key names. To overcome this problem, we introduce the following renaming operator.

Renaming (ρ^c) Renaming is a unary operator that changes a key name or a column name. More specifically, given a collection $\mathfrak{C} =< C, c >$ as input where the schema $C =< k_1 : d_1, k_2 : d_2, ..., k_n : d_n, R >$, a renaming operation is specified as $\rho^c_{a/b}(\mathfrak{C})$ where a and b are attribute names or key names. The result of the renaming operation is a collection $\mathfrak{C}' =< C', c >$ where C' is defined as follows:

- If b is a key name and $b = k_i (1 <= i <= n)$ then $C' =< k_1 : d_1, k_2 : d_2, ..., a : d_i, ..., k_n : d_n, R >$.

- If b is an attribute names then $C' =< k_1 : d_1, k_2 : d_2, ..., k_n : d_n, \rho_{a/b}(R) >$ where $\rho_{a/b}(R)$ replaces the attribute name b with a as defined in the relational algebra.

The renaming operator does not change the collection instance and c' is equal to c. The renaming operator satisfies the following properties:

$$\rho^c_{a/b}(\rho^c_{b/c}(\mathfrak{C})) = \rho^c_{a/c}(\mathfrak{C}) \tag{4.21}$$

$$\rho^c_{a/b}(\rho^c_{c/d}(\mathfrak{C})) = \rho^c_{c/d}(\rho^c_{a/b}(\mathfrak{C})) \tag{4.22}$$

Figure 4.5 illustrates an example of the composition of the renaming and the Cartesian product operators to calculate the Cartesian product of collections M_1 and M_2. By applying the renaming operator the naming conflicts are resolved.

From the above definitions, we can conclude:

Theorem 4.2.9 *The set of collections is closed under union, set difference, selection, projection, Cartesian product, and renaming.*

The proposed collectional operators can be composed arbitrarily to form more complex operations and the result will always be a collection. As an example, given a collection M_1 in Figure 4.2, a scientific query "select all the models whose results are better than the result of model m1" can be expressed as $\pi^c_{Model,Result}(\sigma^c_{Result>=M1.Result}(\rho^c_{M1.Result/Result}(\rho^c_{M1.Model/Model}(\sigma^c_{Model='m1'}(M_1))) \times^c M_1))$. A workflow representation of this query will be shown in Section 3.

We also introduce two operations to modify a collection:

Collection Modification Operators:

- $Insert_{(k_v,v)}(C)$: adds a key-value pair to the collection, if there does not exist a pair (k_v, v') in the relation with the same key value. Otherwise an Union operation is invoked to union v and v'.

- $Delete_{(k_v)}(C)$: removes a pair with a specified key from the collection, if it exists.

Our proposed collectional model is closely related to but differs from the nested relational data model for databases [104]. First, a collection is a set of key-value pairs in which key must be unique and value can be either a relation or another collection structure. The ordered nature and simplicity of collection schema leads to a set of simpler but expressive collectional operators that are amenable to efficient implementation. Second, our collectional model is an ordered model, therefore, our definitions of selection, projection and Cartesian product are dramatically different from their nested relational counterparts. Finally, while the nested relational model is introduced for the storage and querying of structured data, our collectional data model is designed beyond this usage; collections are first-class data objects that are produced and passed from one workflow task to another for further complex computation and analysis, which are not necessarily database operations.

4.3 Collectional Scientific Workflow Composition

We have previously proposed a dataflow-based scientific workflow composition model with composable workflow constructs which are based on a list-oriented data model [53]. In this section, we discuss the application of the collectional model in scientific workflow composition.

In VIEW, a scientific workflow consists of a workflow interface and a workflow body. The workflow interface contains the logical workflow definition which is a tuple $(wid, \mathcal{IP}, \mathcal{OP})$, where *wid* is the unique identifier of the workflow, $\mathcal{IP} = \{i_1, i_2, \cdots, i_m\}$ is the set of input ports, and $\mathcal{OP} = \{o_1, o_2, \cdots, o_n\}$ is the set of output ports. All the inputs and outputs of workflows are required to be collections. The workflow body contains the physical implementation of the workflow.

In our workflow composition model, workflows are the only operands for workflow composition. Tasks such as Web services, Cloud services, local or remote executable programs, are generalized by a task model [86] and constructed as primitive workflows. Since in practice, not all scientific data, especially raw data, will conform to the collectional data model,

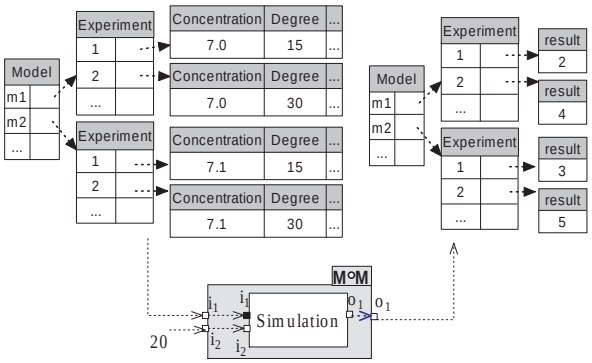


Figure 4.6: The *ParallelSimulation* workflow.

we propose a set of data transformers to convert source data, such as arrays, file collections, and datasets in HDF [5] and NetCDF [8] formats, to collectional data, which can be queried and manipulated by our collectional scientific workflows. For example, an array can be easily transformed to a collection whose key belongs to the domain of natural numbers. Therefore, although the inputs and outputs of tasks can be heterogeneous, they can be casted to collections which are the only data products for workflow processing.

We have proposed a set of workflow constructs, including Map, Reduce, Conditional, and Loop, which are fully composable one with another. Based on the collectional model, we extended four unary workflow constructs to support the collectional model. Below, we illustrate collectional workflow composition by three example workflows taken from the TangoInSilico project.

Parallel processing. Given a *Simulation* workflow, which takes a relation of parameters and an integer indicating the number of experiments as inputs (in this case, 20) and outputs the number of successful matings, Figure 4.6 illustrates the *ParallelSimulation* workflow created by applying the composition of two Map constructs on the *Simulation* workflow. The *ParallelSimulation* workflow takes a collection of parameters as input, and executes the *Simulation* workflow for each set of parameters in parallel. The collectional model supports the same nested parallelism as the list model. However, in contrast to the list model, which uses

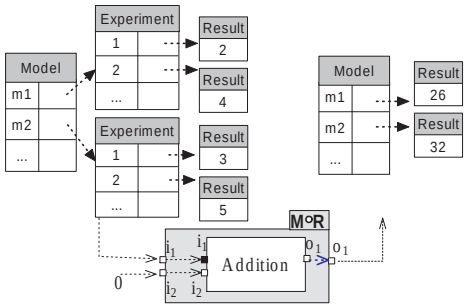


Figure 4.7: The *ParallelAggregation* workflow.

only integer indices, the collectional model uses explicit keys for indexing. This provides a more meaningful hierarchical organization classified by *Model* and *Experiment*. Furthermore, the collectional model uses relations to represent table-like parameters. As a matter of fact, the list model is a special case of the collectional model where the key values are integers.

Parallel aggregation. Figure 4.7 illustrates the *ParallelAggregation* workflow created by applying the composition of the Map and Reduce constructs on an *Addition* workflow. The resultant workflow takes the output of the *ParallelSimulation* workflow, and aggregates the results for each model. Because of the set-oriented nature of collectional model, the aggregation order is often not important. We propose a parallel version of the Reduce construct which is implemented by tree-like parallel aggregations (a binary tree in this example is shown in Figure 4.8.)

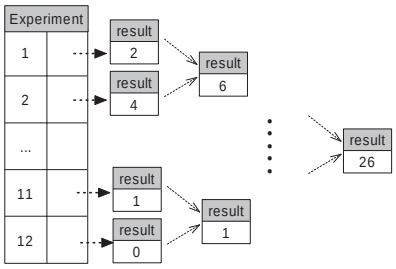


Figure 4.8: An example of the parallel Reduce construct.

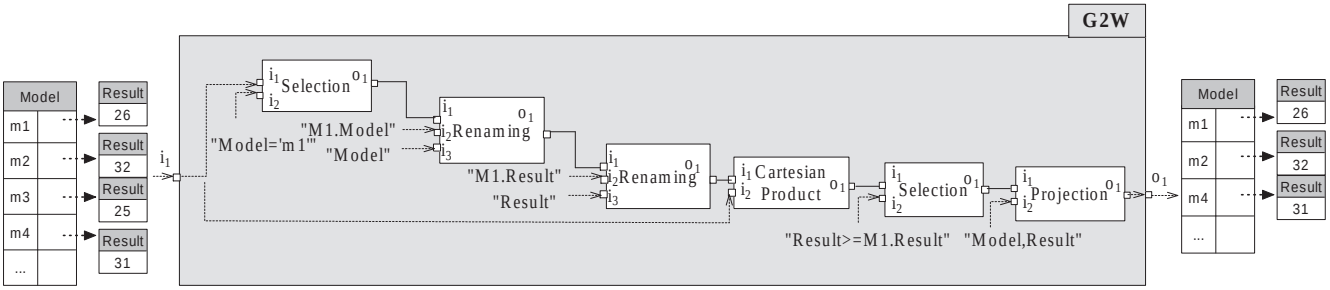


Figure 4.9: The *Query* workflow.

Collectional Query. Figure 4.9 illustrates the *Query* workflow that executes a scientific query "select all the models whose results are better than the result of model m1" as introduced in Section 2. The *Query* workflow is created by applying a G2W construct (a construct to map a workflow graph to a workflow) on a workflow graph. The workflow graph consists of several primitive workflows that implement collectional operators.

4.4 Chapter Summary

In this chapter, we formalized a collectional data model as the basis for a scientific workflow composition framework. Our method seamlessly leverages the advantages of the relational model and database techniques into scientific workflows. Moreover, our collectional model extends the relational model to manage hierarchically structured collections of scientific datasets.

CHAPTER 5

VIEW: A PROTOTYPICAL SCIENTIFIC WORKFLOW MANAGEMENT SYSTEM

Having defined a scientific workflow composition model and a collectional data model, this chapter presents the design and implementation of the proposed techniques in a new version of the VIEW system [85]. The development of the VIEW system complies with the principle of *minimum complexity for users, but massive techniques in the backstage*. The increasing use of scientific workflow systems correlates with the simplicity of the workflow paradigm that provides a clear and simple abstraction for manipulating and coordinating resources. Scientific workflow techniques are proposed to facilitate scientists and allow them to concentrate on their research at the problem domain level without requiring deep knowledge of programming languages, operating systems, or hardware infrastructure. The remainder of this chapter is structured as follows. Section 5.1 illustrates the service oriented architecture of the VIEW system. Section 5.2 illustrates the implementation of the workflow engine. Section 5.3 covers the implementation of the collectional data model in the data product manager and Section 5.4 elaborates data typing in VIEW. Section 5.5 describes predefined data operators, in the form of primitive workflows, and presents an example of their composition to query relations and collections. Section 5.6 summarizes this chapter.

5.1 VIEW *Architecture*

The VIEW system implements the service-oriented reference architecture proposed in [85]. Figure 5.1 presents the overall architecture of the VIEW system consisting of six autonomous, reusable, and independent service components. Other than the workbench, each subsystem is exposed with Web services and defines its functional interface in WSDL: I_{WE}, I_{WM}, I_{TM},

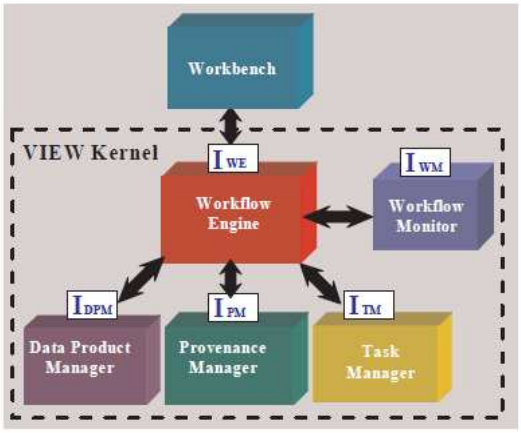


Figure 5.1: Overall architecture of the VIEW system [85].

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. The *workbench* subsystem is responsible for the design of scientific workflows, the presentation of data product and data provenance information, as well as the system status. Workbench consists of a workflow design panel and a system management panel. The workflow design panel allows the users to drag and drop existing workflows and compose them with workflow constructs to formalize new composite workflows. The workbench will automatically translate the graphical composition into a workflow definition file, and then send it to the workflow engine to register the new workflow. The system management panel contains work spaces of *workflow management*, *data product management*, *provenance management* and *workflow runtime management*. Each work space correlates to a subsystem and VIEW allows users to dynamically select and configure the services of each subsystems. The workflow management work space allows users to create workflows and to browse, search, manage, execute, and reuse existing workflows. The data product management work space allows users to create data products, and to browse, search, manage, and use existing data products. The provenance management work space allows users to query and

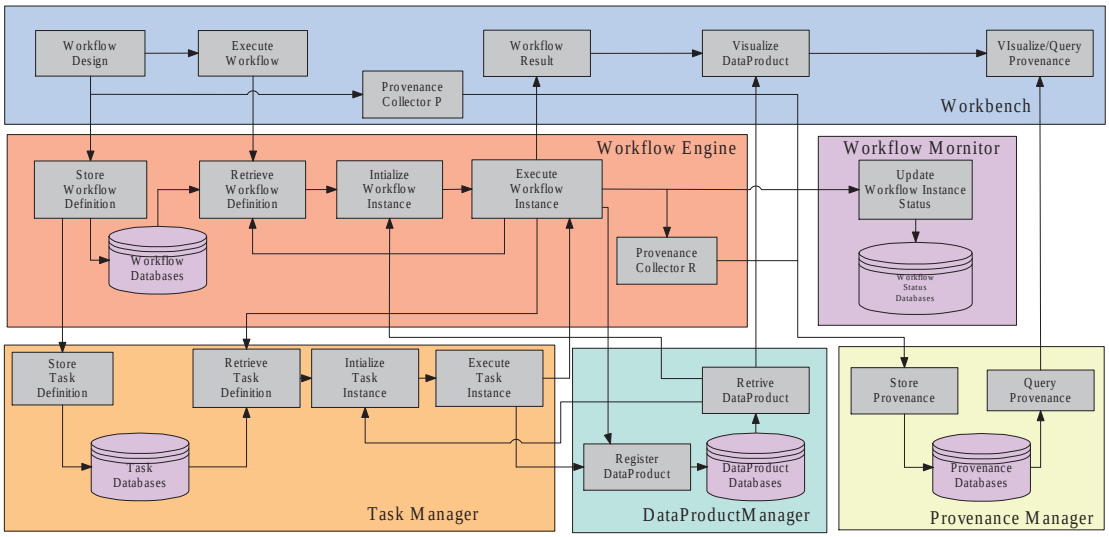


Figure 5.2: A typical scientific workflow execution diagram.

visualize the provenance information of previous workflow executions. Finally, the workflow runtime management work space reports the statuses of current running workflows.

Figure 5.2 illustrates a diagram of typical scientific workflow execution. Firstly, scientists design a graphical workflow composition in the workbench which will be recorded automatically into a workflow definition file following the SWL language. The workflow definition will then be sent to the workflow engine and stored in databases. After the workflow is registered, scientists are able to execute it providing a set of input data set. In VIEW, all scientific data are managed by the data product manager and the workflow engine transfers only virtual data products which are references. The workflow engine supports multi-users. For each workflow execution request, workflow engine will initialize *an instance of the workflow*, bind the input data to workflow input ports, and create a thread to execute this workflow instance. The runtime information is monitored by the workflow monitor and a provenance file will be generated based on the OPM model [95] [82] [83] and sent to the provenance manager after each workflow execution. Workflows are executed according to their implementations: primitive workflows will invoke the encapsulated task components by sending requests to the task

Figure 5.3: (a) A SWL specification example of the `workflowInterface` definition of a unary-construct-based workflow. (b) a SWL specification example of the `workflowBody` for graph-based workflow (b-4), primitive workflow (b-1), and unary-construct-based workflow (b-2); (b-3) a SWL specification example of the `workflowBody` definition for unary-construct-based workflow with a composition of the Map construct and the Reduce construct; (b-5) a SWL specification example of the exception handling.

manager; composite workflows will decompose the workflow construction and recursively invoke sub-workflows. After the workflow execution, workflow outputs will be returned to the workbench which contains references of the final data products. Users can then retrieve data products from the data product manager and visualize data products with built-in or specified visualization tools. Moreover, users can also visualize and query provenance using the OPQL language [84]. The architectural design of the workflow engine, the task manager, and the data product manager will be elaborated in the following sections.

5.2 Workflow Engine

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. It realizes the proposed workflow model of Chapter 3 into an XML-based scientific workflow specification language (SWL). The XML schema of SWL is shown in Appendix A. A SWL specification

consists of a set of workflows including one root workflow as an entry point for the workflow engine. As shown in Figure 5.3.(a), each workflow definition is clearly separated into the logical layer and the physical layer: `workflowInterface` and `workflowBody`. The `WorkflowInterface` element contains subelements `inputPorts` and `outputPorts` to specify the input and output ports of the workflow. The `workflowBody` element defines the functional body of the workflow which has three types: primitive, graph-based and unary-construct-based.

Figure 5.3.(b-1) illustrates an example of primitive `workflowBody` which contains subelements `taskComponent` and `T2W`, which defines the invocation methods of a task component and input/output mappings. The details of the task definition will be described later in this section.

Figure 5.3.(b-4) illustrates an example of graph-based `workflowBody` which contains subelements `workflowGraph` and `G2W`. The `workflowGraph` element specifies a workflow graph consisting of a set of workflow instances (by the `workflowInstances` element) and a set of data channels (by the `dataChannels` element). The `G2W` element defines the input/output mapping between the workflow graph and the workflow (by the `inputMapping` and `outputMapping` elements).

Figure 5.3.(b-2) illustrates an example of unary-construct-based `workflowBody` which contains subelements `baseWorkflow` and `unary-construct`. The `baseWorkflow` element refers to the workflow to be constructed and the `unary-construct` element contains either a unary construct (as shown in Figure 5.3.(b-1), the `map` element specifies the Map construct) or a composition of unary constructs (as shown in Figure 5.3.(b-2), the `reduce` and `map` elements specify a composition of Map and Reduce). SWL hides a default mapping between the unary-construct-based workflow and its base workflow. The unary-construct-based workflow will contain the same set of input/output ports as the base workflow; the designated ports are specified in the unary construct element (e.g., the `mapport`

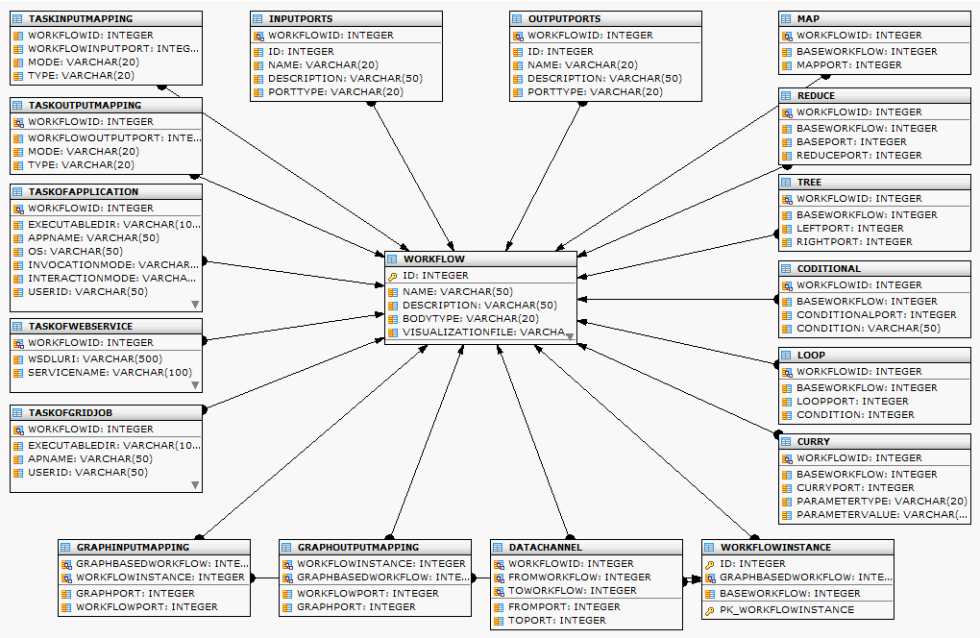


Figure 5.4: Relational database schema for our scientific workflow composition model.

attribute in the `map` element) and the type of the designated ports might be changed implicitly according to the semantics of the construct. Figure 5.3.(b-5) illustrates an example of the exception construct which specifies the port to detect, the predicate and the exception code.

SWL is also realized in a relational database schema as shown in Figure 5.4. The *WORKFLOW* table holds the general information including the workflow identifier, name and description. The *INPUTPORTS* and *OUTPUTPORTS* tables hold the information of the workflow interface and include a foreign key *workflowID* to relate ports to corresponding workflows. The detailed task component definition is stored in three tables including *TASKOFWEBSERVICES*, *TASKOFAPPLICATION*, and *TASKOFGRIDJOB*, which store the definition of three different type of task components: Web Services, local or remote executables applications, and Grid jobs, respectively. The *TASKINPUTMAPPING* and *TASKOUTPUTMAPPING* tables stores the input/output mappings between the workflow and the task. The *MAP*, *REDUCE*, *TREE*, *CONDITIONAL*, *LOOP*, and *CURRY* tables store the definitions of the corresponding unary constructs. Because a unary construct defines a one-to-one mapping,

the input/output mappings are by default and abbreviated. The *NODEWORKFLOW* and *DATALINK* tables define the workflows and data link constructs that constitute the workflow graphs. The *GRAPHINPUTMAPPING* and *GRAPHOUTPUTMAPPING* tables define input/output mapping between the workflow graph and graph based workflow.

The workflow engine interface is defined in WSDL including five Web service operations:

- *RegisterWorkflow* registers a workflow with a workflow definition in SWL.
- *GetWorkflow* returns the definition of a workflow.
- *DeleteWorkflow* deletes a workflow.
- *ExecuteWorkflow* executes a workflow with given inputs.
- *GetWorkflowList* returns a list of existing workflow information for browsing.

The WDSL specification of the *Workflow Engine* is shown in Appendix C.

Upon a request of workflow execution, the workflow engine will first retrieve the workflow definition from the database and create a runtime *workflow instance*. A workflow instance is a *new copy* of the workflow body biding with given workflow inputs, rather then updating the original body. This is necessary because a workflow may be executed multiple times with different inputs, and its definition serves as a "template", from which an instance is constructed each time it is executed. The template itself should not be changed during any execution process. A workflow instance is *executable* if every input port is bound with a data product.

A executable workflow instance can be executed according to its workflow body type:

Primitive workflow body. Primitive workflows are basic building block of workflow composition. A primitive workflow encapsulates a mapping from a task component that can be heterogeneous and distributed. While in our original design, the task manager implements a task model [86], in which a task consists of a task template and a task component, resulting

in a tedious mapping from workflow to task template and then to task component; in the new version of the VIEW system, we integrated task model into our workflow model by a direct mapping from a primitive workflow to the corresponding task component. Furthermore, this mapping can now be done by the system automatically. When registering a task, the user specifies the target task component, invocation method, and the task inputs/outputs, and the system will create a primitive workflow to encapsulate this task component and the mappings between the workflow and the task component are automatically generated and stores in the workflow specification. Figure 5.5 illustrates an example specification of a primitive workflow expanded from Figure 5.3.(b-3), which contains the mapping from a Windows application. The *taskType* attribute defines the type of the task component. Current task manager supports three types of task: "WebService", "WindowsApplication", and "GridJob". The task types are extensible in both the language and the system. The *executable* element and the *appName* element specify the location and the name of the target Windows application. The *taskInvocation* element specifies the task invocation method including the *operatingSystem* element that specifies the operating system such as windows and unix; the *invocationMode* element that specifies whether the task is local or remote; and the *interactionMode* that specifies whether it is a user-interactive task or not. A user-interactive task can only been invoked in the workbench side to interact with users directly. The *T2W* element defines the mapping between the task component and the workflow (by the *inputs* and *outputs* elements). The task manager will use those information to bind and transfer data from workflow to task components.

When executing a primitive workflow, the workflow engine will extract the workflow definition and send it to the task manager. The details of the execution of the heterogenous task components can be found in [86]. The current task manager subsystem is responsible only for the execution of task components. The task manager is separated from the workflow engine in order to reduce the work load of the workflow engine so that it will not become a bottleneck.

```
<workflow root="true" name ="MeshHoleFill">
  <workflowInterface>
    <workflowDescription>Fill holes in the iso-surface.</workflowDescription>
    <inputPorts nunber="2">
      <inputPort id = "i1">
        <portName>isoSurfaceFile</portName>
        <portType>File</portType>
      </inputPort>
      <inputPort id = "i2">...</inputPort>
      <inputPort id = "i3">...</inputPort>
    </inputPorts>
    <outputPorts nunber="1">
      <outputPort id = "o1">
        <portName>holesCoveredFile</portName>
        <portType>File</portType>
      </outputPort>
      <outputPort id = "o2">...</outputPort>
    </outputPorts>
  </workflowInterface>
  <workflowBody type="primitive">
    <taskComponent taskType="WindowsApplication">
      <executable>file://localhost/OBJFILL.exe </executable>
      <appName>OBJFILL</appName>
      <taskInvocation>
        <operatingSystem>Windows</operatingSystem>
        <invocationMode>Local</invocationMode>
        <interactionMode>No</interactionMode>
      </taskInvocation>
    </taskComponent >
    <T2W>
      <inputs>
        <input id="i1" mode="File" name = "/OBJFILL.obj" type ="FILE(OBJ)" / >
        <input id="i2" mode="EnvironmentVariable" name = "inputEnv" type ="String" / >
        <input id="i3" mode="ConstantCLArg" name = "inputCLArg" type ="String" / >
        </inputs>
      <outputs>
        <output id="o1" mode="File" name = "Subjhfobj" type ="FILE(OBJ)" / >
        <output id="o2" mode="ExitReturnValue" name = "inputEnv" type ="Integer" / >
      </outputs>
    </T2W>
  <workflowBody>
</workflow>
```

Figure 5.5: An example specification of a primitive workflow.

Unary-construct-based workflow body. a unary-construct-based workflow contains one of the Map, Reduce, Tree, Conditional, Loop, and Curry constructs, or a composition of those constructs. The execution of a unary-construct-based workflow is based on the semantics introduced in Chapter 3.

Graph-based Workflow body. Execute the workflow graph. A workflow graph is a tuple (W, C) where W is a set of workflow instances and C is a set of data channels. VIEW applies a dynamic and parallel scheduling algorithm. First, the workflow engine will bind input data to the corresponding input ports of the specified workflow instances according to the graph

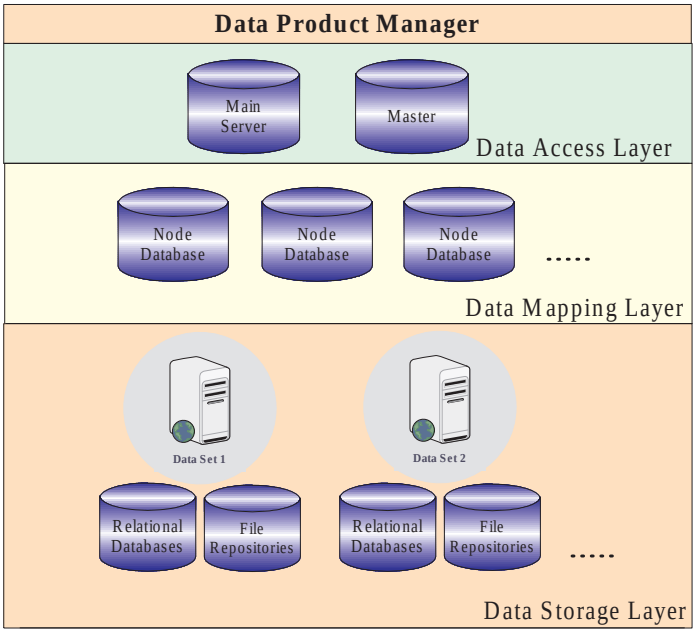


Figure 5.6: Architecture of the data product manager.

input mapping definition. The workflow engine will iterate over the set of workflow instances and create a new thread for each executable workflow instance. Once a workflow instance execution finishes successfully, the generated output data products will be bound to the corresponding output ports, and then sent to linked input ports of other workflow instances or output mapping via data channels. Then workflow engine will then check whether those data products makes other workflow instances executable, and if so, execute them. The execution of a workflow graph finishes successfully if every output port is bound with a data product, and the execution of a workflow graph aborts if either all workflow instances finished and there is at least one output port is not bound with a data product, or there is no running workflow instances and no executable workflow instances.

5.3 Data Product Manager

5.3.1 Architecture of the Data Product Manager

The *data product manager* subsystem is responsible for the storage and the management of distributed collectional scientific data. Figure 5.6 illustrates the three-layer architecture of the data product manager.

The first layer is *the data access layer* which provides the access interface. A main server contains a root table which stores general information of all data products, e.g. name, description, and their locations in slave databases. A master server maintains access methods of all slave databases. Upon a request to retrieve a data product, the data product manager will first search the root table to find the data product location in the slave server and then retrieve it.

The second layer is *the data mapping layer* which provides a mapping from the logical data model to the physical organization. This layer consists of a series of node databases. While the collectional data model is logically hierarchically structured, the physical storage can but not necessarily be hierarchically organized in the same way. Current VIEW system supports two ways of storage. First, a collection can be stored in a table containing a set of its key/value pairs, whose values are references to existing collections. Such implementation follows its logical organizations. Second, a collection can be expanded and physically separated into two parts: the structural metadata and the relations. The structural metadata of a collection is stored in one table including key values and access methods of the relations. The relations are distributively stored in the third layer. Below we define two operators to enable the second type of storage.

Compress(ϱ): The *Compress* is a unary operator that reduces the number of keys of the collection while increases the number of columns of inner relations accordingly. More specifically, given a collection $\mathfrak{C} =< C, c >$ as input with schema $C =< k_1 : d_1, k_2 : d_2, ..., k_n :$

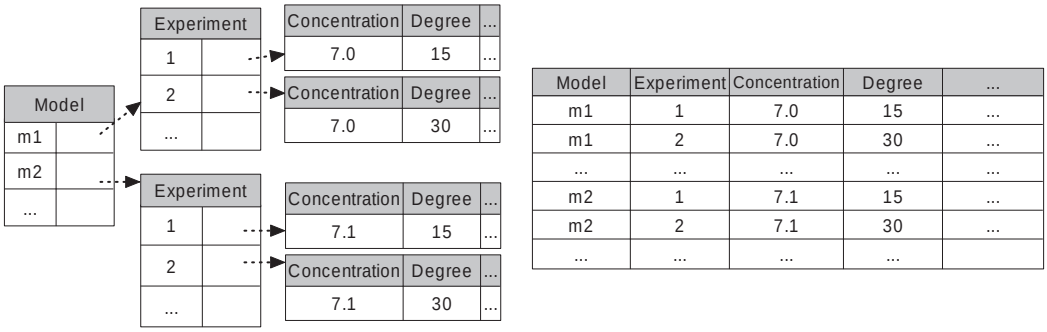


Figure 5.7: Example of the Compress operator: (a) the original relation *Parameters*; (b) The result collection *RParameters* from the operation $\varrho(\varrho(Parameters))$.

$d_n, R >$, a *Compress* operation can be specified as $\varrho(\mathfrak{C})$. The result of the *Compress* operation can be either a collection or a relation depending on the height of the input collection, more formally, the resultant schema is defined as follows:

- If $H(C) > 1$, then the Compress operation returns a collection $\mathfrak{C} =< C', c >$ where $C' =< k_1 : d_1, k_2 : d_2, ..., k_{n-1} : d_{n-1}, (k_n : d_n) \times R >$.

- If $H(C) = 1$, then the Compress operation returns a relation $\mathfrak{R} =< R', r >$ where $R' = (k_1 : d_1) \times R$.

We then define the Compress at the instance level as follows:

$$\begin{array}{ll} \varrho(c) = \{(p, \varrho(q)) : (p, q) \in C\} & if(H(C) > 1) \\ \cup_{(p,q)\in c} (p \times^{rc} q) & if(H(C) = 1) \end{array} \tag{5.1}$$

The Compress operator can be used to transform collections to relations. For example, Figure 5.7 illustrates an example of the Compress operator, in which the relation *RParameters* is obtained by the operation $\varrho(\varrho(Parameters))$.

Group By(τ): The *Group By* is a reverse operator of the *Compress* operator which takes either a relation or a collection as input and returns a collection. On the one hand, given a

collection $\mathfrak{C} =< C, c >$ as input with the schema $C =< k_1 : d_1, k_2 : d_2, ..., k_n : d_n, R >$ and $R =< c_1 : o_1, c_2 : o_2, ...c_m : o_m >$, a Group By operation can be specified as $\tau_{c_l}(\mathfrak{C})$ where $c_l \in \{c_1, c_2, ..., c_m\}$. The result of the Group By operation is a collection $\mathfrak{C}' =< C', c' >$ where $C' =< k_1 : d_1, k_2 : d_2, ..., k_n : d_n, c_l : o_l, \pi_{c_1,...c_{l-1},c_{l+1},...c_m} R >$, and

$$
\begin{aligned}
& c' = \\
& \{(p, \tau_{c_l}(q)) : (p, q) \in c\} \qquad\qquad if(\mathcal{H}(C) > 1) \\
& \{(p, \{(k, \pi_{c_1,...c_{l-1},c_{l+1},...c_m}(\sigma_{c_l=k}(q))) : k \in \pi_{c_l}(q)\}) : \\
& \quad (p, q) \in c\} \qquad\qquad if(\mathcal{H}(C) = 1)
\end{aligned}
\tag{5.2}
$$

On the other hand, given a relation $\mathfrak{R} =< R, r >$ as input where the schema $R =< c_1 : o_1, c_2 : o_2, ...c_m : o_m >$, a *Group By* operation can be specified as $\tau_{c_l}(\mathfrak{R})$ which transforms the relation to a collection $\mathfrak{C}' =< C', c' >$ where $C' =< c_l : o_l, \pi_{c_1,...c_{l-1},c_{l+1},...c_m} R >$, and

$$
c' = \{(k, \sigma_{c_r=k}(\pi_{c_1,...c_{r-1},c_{r+1},...c_m}(r))) : k \in \pi_{c_r}(r)\} \tag{5.3}
$$

Given any collection $\mathfrak{C} =< C, c >$, the *Compress* operation and the *Group By* operation are reversible:

$$
\tau_k(\varrho(\mathfrak{C})) = \mathfrak{C} \quad where \quad k = \mathcal{K}_{\mathcal{H}(\mathfrak{C})} \tag{5.4}
$$

As an example, the $\tau_{Codition}(\tau_{Model}(\textit{RParameters}))$ operation will return the *Parameters* collection.

The Compress and Group By operators provide foundations for the lossless mapping between the collectional model and the relational model. Therefore, we are able to expand a collection to a relation and store all the key values in one table.

Finally, the third layer is *the data storage layer* which includes distributed relational data. Relational data are not necessarily stored in data product manager but can be distributed remotely as long as it provide well defined accessing methods. Our relational model is relaxed

```
<dataProduct name ="Parameters">
   <description>A collection of parameters.</description>
   <data>
     <collection>
        <collectionalSchema>
          <key name="Experiment" type="Integer"/>
        </collectionalSchema>
        <collectionalInstance count=count>
          <pair>
            <key>1 </key>
            <relation>
              <relationalSchema>
                <column name="speed" type="Decimal"/>
                <column name="angle" type="Integer"/>
              </relationalSchema>
              <relationalInstance>
                <row>
                  <speed>50</speed>
                  <angle>0</angle>
                </row>
                ...
              <relationalInstance>
            </relation>
          </pair>
          ...
        </collectionalInstance>
      <collection>
   </data>
</dataProduct>
```

Figure 5.8: Example of the XML description of a collectional data product.

to support files which are widely used in many scientific applications and Grids, In addition to the traditional *BLOB* type, our system defines a *FILE* type containing a reference to a local/remote file. Therefore, large files do not need to be physically stored into databases.

5.3.2 Interface of the Data Product Manager

We also propose an XML-based data product specification language (DPL) to enable the transfer of collectional data products. The XML schema of DPL is shown in Appendix B. Figure 5.8 shows an example of the DPL description of a collectional data product. The *collecition* element stores the collectional data including the collectional schema and the instance of that schema. The *collectionalSchema* element specifies all enclosed key names and types. The *collectionalInstance* tag contains the key-value pairs. This example illustrates a collection with Height of 1 and the values are relations. The *relation* element also includes

the relational schema and the instance. The *relational schema* elements defines column names and column types while the *relationalInstance* contains a set of rows.

The data product manager provides the following three Web service operations?:

- *RegisterDataProduct* registers a data product with a data product definition.
- *GetDataProduct* returns a data product.
- *DeleteDataProduct* deletes a data product.
- *GetDataProductList* returns a list of existing data product information for browsing.

The WDSL specification of the *Data Product Manager* is shown in Appendix D.

5.4 Data Type System in VIEW

While the task manager separates the physical execution of heterogeneous task resources from logical workflow composition and orchestration, the data product manager separates the physical data management and transfer from the logical dataflow transition. Such separation not only conceals the implementation details but also reduces unnecessary physical data movements.

Following the definitions in the second chapter, each scientific workflow is associated with a set of input ports I and a set of output ports O. Each port p is associated with a data domain specifying the type of data product that the port can accept. VIEW is a dataflow-based scientific workflow management system and data movement in VIEW are modeled as dataflows. A dataflow transition represents a transfer of dataflow from an output port to an input port via a data channel. A data channel is a tuple (o, i) where o is an output port and i is an input port.

The VIEW system conforms to the collectional data model. Each data product is a collection and the data type is determined by its *schema*. First, a set of scalar domains are defined. Table 5.1 summarizes scalar data domain types in view, as well as their mappings to MySQL,

Table 5.1: Scalar data type mappings among VIEW, MySQL, and XML.

VIEW DataType	MYSQL DataType	XML DataType
STRING	VARCHAR	string
INTEGER	INTEGER	int
LONG	BIGINT	long
DECIMAL	DECIMAL	decimal
FLOAT	FLOAT	float
DOUBLE	DOUBLE	double
BOOLEAN	BOOLEAN	Boolean
DATETIME	DATETIME	dateTime
BLOB	LONGBLOB	base64Binary
FILE	-	-

and XML data types. Other databases including Oracle, Microsoft SQL Server, and Firebird are also supported in VIEW.

In order to reduce trivial data conversions while guaranteeing a strict data typing. we define the following rules to facilitate safe and automatic transformation.

Definition 5.4.1 *A scalar data type t_1 is* superior *to a scalar data type t_2, denoted as $t_1 \succ t_2$, iff they satisfy one of the following conditions:*

- $t_1 = t_2$
- $t_1 = STRING$ *and* t_2 *is any atomic data type.*
- $t_1 = LONG$ *and* $t_2 = INTEGER$.
- $t_1 = DECIMAL$ *and* $t_2 \in INTEGER, LONG, FLOAT, DOUBLE$.
- $t_1 = DOUBLE$ *and* $t_2 \in INTEGER, LONG, FLOAT$.
- $t_1 = FLOAT$ *and* $t_2 \in INTEGER, LONG$.

Then we define the type comparison between two general collection schemas.

Definition 5.4.2 *A relation schema* $R_1 =< c_{11} : d_{11}, ..., c_{1n} : d_{1n} >$ *is* superior *to a relation schema* $R_2 =< c_{21} : d_{21}, ..., c_{2n} : d_{2n} >$*, denoted as* $R_1 \succ R_2$*, iff* $c_{11} = c_{21}, ..., c_{1n} = c_{2n}$*, and* $d_{11} \succ d_{21}, ..., d_{1n} \succ d_{2n}$.

Definition 5.4.3 *A collection schema* $C_1 =< k_1 : d_1, S_1 >$ *is* superior *to a collection schema* $C_2 =< k_2 : d_2, S_2 >$*, denoted as* $C_1 \succ C_2$*, iff* $k_1 = k_2$*,* $d_1 \succ d_2$*, and* $S_1 \succ S_2$.

While the data product manager encapsulated the collectional data model, in the workflow engine, a collection is treated as an *atom* including an identifier and a type which is its collectional schema. A *List* data construct is then introduced to facilitate the Map, Reduce, and Tree constructs. More formally, in the workflow engine, a *data product* d is either an *atom*, a list $< d_1, ...d_n >$ whose elements d_i are data products, or ϵ, an empty data product.

Definition 5.4.4 (Data Product Type) We define the type of a data product d, denoted as $\mathfrak{T}(d)$, as follows:

- $\mathfrak{T}(d) = C$, if d is an atom representing a collection $\mathfrak{C}(C, c)$ and C is collectional schema.
- $\mathfrak{T}(d) = L$, if d is an List and L is a primitive list type (VIEW allows heterogeneous lists).
- $\mathfrak{T}(d) = \epsilon$, if d is an ϵ.

$\diamondsuit$

The list type is only introduced in the workflow engine mainly to support the Map, Reduce and Tree construct, and the only primitive workflow that accept lists are two built-in workflows $Merge$ and $Split$ introduced in Chapter 3. The list type is not exposed to application tasks. Instead, they can use collections which do not allow heterogeneous values but are better organized and provide a rich set of operators.

When registering a workflow, the workflow engine will validate each data link in graph-based workflows by checking types of the linked ports, denoted as $\mathfrak{T}(p)$ where p is a port.

Definition 5.4.5 *A data channel* (o, i) *is* valid *iff* $\mathfrak{T}(i) \succ \mathfrak{T}(o)$.

Current data type system focuses only on data modeling. However, it is extensible for semantic type check if a domain ontology is well defined. We are also investigating the possibility of modifying our workflow constructs and integrating the workflow model and the collectional data model, which will be discussed in the future works.

5.5 Scientific Workflow approach for Collectional Data Querying

Scientific workflow technologies enable visual composition of queries instead of traditional query languages such as SQL. The VIEW system provides a set of built-in primitive workflows as follows:

Arithmetic Operators:

- Addition: takes inputs of two numeric types, returns their sum.
- Substraction: takes inputs of two numeric types, returns their difference.
- Multiplication: takes inputs of two numeric types, returns their product.
- Division: takes inputs of two numeric types, returns their quotient.

Boolean Operators:

- Add: takes inputs of two boolean types, returns their conjunction.
- Or: takes inputs of two boolean types, returns their disjunction.
- Not: takes input of a boolean type, returns its negation.

Collectional Operators:

- Selection: takes input of a collection and select condition, returns the selected collection.
- Projection: takes input of a collection and project keys and columns, returns the projected collection.
- Union: takes input of two collections, return their union.
- Difference: takes input of two collections, return their difference.
- Cartesian Product: takes input of two collections, return their Cartesian product.
- Rename: takes input of a collection, the column/key to be renamed, and the new name, returns the renamed collection.
- Natural Join: takes input of two collections, return their nature join.
- Join: takes input of two collections, and join condition, return their join.
- Outer joins (Left Join, Right Join, Full Join): takes input of two collections, and join condition, return their out joins.
- Group By: takes input of a collection and the column to be grouped, returns the result collection whose height increased.
- Compress: takes input of a collection and the key to be compress, returns the result collection whose height decreased.
- Order By: takes input of a collection and the column/key to be ordered, returns an ordered collection.
- Aggregation Operators(e.g. Sum, Avg, Max, Min, Count): takes input of a collection, and the key/column to be aggregated, returns aggregated result.

List Operators:

- Merge: takes any number of inputs of any types, returns a list of those inputs, ordered by the port id.

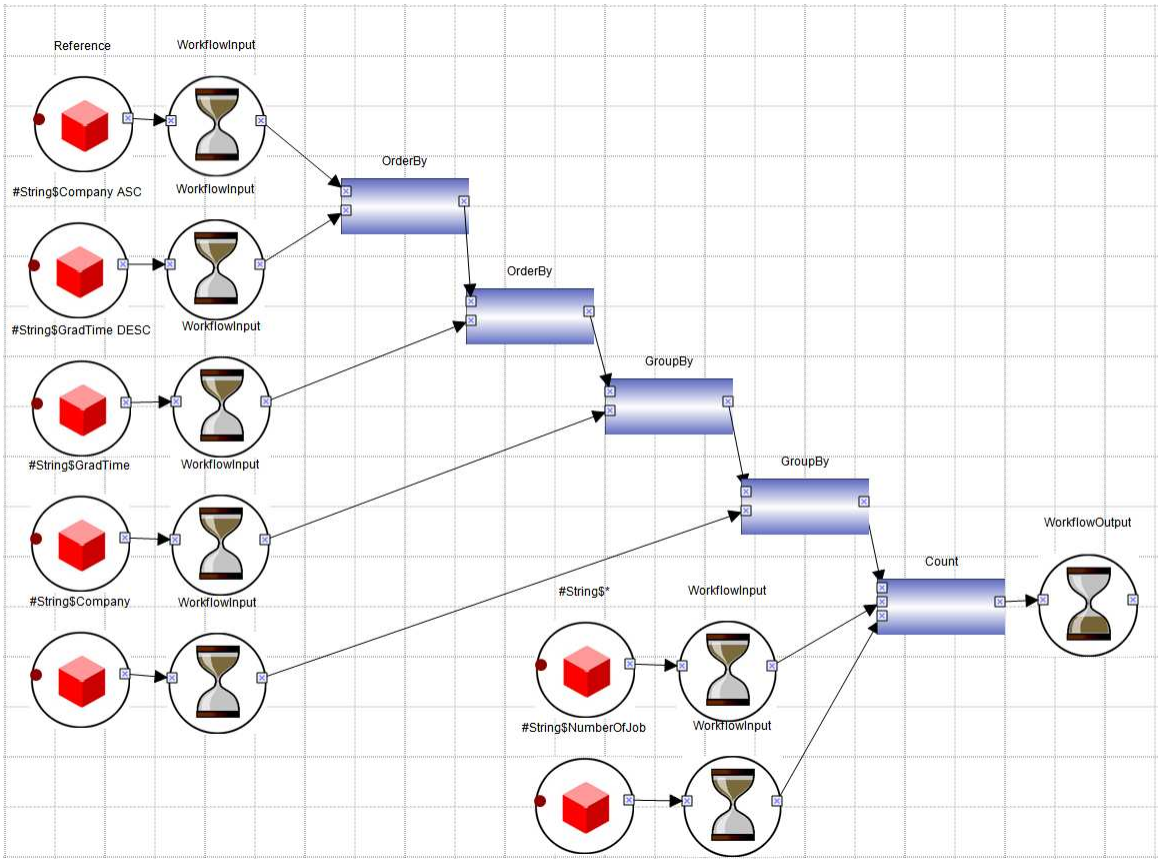


Figure 5.9: Example of a query workflow.

- Split: takes input of a list, split it and returns each internal data product to a port one by one.

Those operators can be composed into scientific workflows to implement arbitrary queries. As an example, given a table $Reference < Student, Company, GradTime >$, which stores graduated students' current company and their graduation years, a query "Find the total number of students offered in each company and each graduation year; Sort the result in descending-GradTime and ascending-Company order." can be implemented in the following the SQL query:

SELECT Company, GradTime, COUNT(DISTINCT Student) AS NumberOfJob

```
FROM Reference
GROUP BY Company, GradTime
ORDER BY GradTime DESC, Company ASC;
```

While the GROUP BY statement is introduced in SQL to in conjunction with the aggregate functions to group the result-set by one or more columns, itself is not semantic complete because the result of the GROUP BY statement is a set of tables (actually a collection) which can not be modeled in the relational model. However, it is elegantly supported in our collectional model and any collectional operators including the aggregate operators can be applied on its result. Figure 5.9 illustrates a query workflow which is equivalent to the above query. By applying the *Group By* operator twice, we get a collection with two keys: Company and GradTime. The following *Count* operator counts the number of students in each company and each graduation year.

5.6 Chapter Summary

This chapter illustrated the architectural design of the VIEW system and the detailed implementation of the workflow engine, and the data product manager. The VIEW system provides an integrated platform with a simple and user-friendly interface for domain scientists to perform in silico scientific experiments and systematic scientific data analysis. Besides the case studies presented in previous chapters, we also show in this chapter the ability of the VIEW system to support relational and collectional queries with our data model and well defined operators.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

We are in the beginning of a era of "e-science". Advances in computational technology are transforming discovery and research in nearly all scientific fields. In the meantime, the coupling between scientists and computer technology also created an explosion in computation and data volume, and leads to increased user needs and expectations for new tools. These trends give rise to many new problems and opportunities in various domains, e.g. more scalable method for data management, more efficient algorithms for data analysis, more advanced technology for visualization, and more importantly, a more powerful and mature platform to integrate those multi-disciplinary techniques. This thesis uses the scientific workflow approach to address such need.

In this thesis, we first proposed a new dataflow-based scientific workflow composition model. My contributions include: 1.) seven key requirements for a scientific workflow composition model based on a comprehensive literature review and our experience in developing the VIEW system; 2.) a new scientific workflow model which separates the declaration of the workflow interface from the definition of its functional body; 3.) a set of workflow constructs, including Map, Reduce, Tree, Loop, Conditional, and Curry, which are fully compositional one with another; and 4.) a dataflow based exception handling approach to support hierarchical exception propagation and user-defined exception handling.

In order to support hierarchical collection-oriented scientific data, we proposed a collectional model including a collection structure to model hierarchical collection-oriented scientific data, and and a set of operators to manipulate and query such data. To our best knowledge, this is the first algebraic approach to modeling collection-oriented scientific data.

We realized proposed techniques in the VIEW system. The VIEW system is designed and implemented using service oriented architecture following the reference architecture of SWFMSs. VIEW comprises six loosely coupled subsystems: a workbench to visually design workflows, a workflow engine to manage and execute workflows, a task manager to execute tasks, a workflow monitor to display system status and track exceptions, a provenance manager to store and query workflow provenance, and a data product manager to store and manage data products. Each system implements its own model independently of the other systems. This thesis presented the design and the implementation of the workflow engine, the task manager, and the data product manager, as well as the coordination and integration of these subsstems.

In the future, scientific workflow will become one of the key techniques to organize large-scale scientific projects which are becoming more and more computation intensive and data intensive. My future work focus typically on the following topics:

Formalization of the scientific workflow algebra and the collectional algebra Although we have formally defined a scientific workflow composition model and a collectional data model, there is still much work to be done in order to formalize the scientific workflow algebra and the collectional algebra. We plan to summarize the properties of operators and research the completeness of our proposed models. We also plan to compare our models to other related workflow and data models and explore the possibility of the commutation or even integration between different systems.

Collaborative scientific workflow composition Collaborative scientific workflow composition has recently been proposed to support collaborative scientific research projects, which require intensive collaboration among scientists with diverse expertise. A collaborative scientific workflow management system allows participating scientists to design and compose common scientific workflows concurrently. This poses challenges to the architectural design, and the consistency of the scientific workflow management systems. In order to

solve those challenges, we plan to borrow the locking and transaction processing techniques from Database and apply them to support collaborative scientific workflows.

APPENDIX A

Scientific Workflow Language (SWL)

<?xml version=”1.0” encoding=”UTF-8”?>

<xsd:schema xmlns:xsd=”http://www.w3.org/2001/XMLSchema” elementFormDefault=”qualified” attributeFormDefault=”unqualified”>

<xsd:element name=”workflowSpec”>

<xsd:complexType>

<xsd:sequence>

<xsd:element name=”workflow” type=”WorkflowXMLElementType” maxOccurs=”unbounded”/>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

<xsd:complexType name=”WorkflowXMLElementType”>

<xsd:sequence>

<xsd:element name=”workflowInterface”>

<xsd:complexType>

<xsd:sequence>

<xsd:element name=”workflowDescription” type=”xsd:string” minOccurs=”0”/>

<xsd:element name=”inputPorts”>

<xsd:complexType>

<xsd:sequence>

<xsd:element name=”inputPort” type=”PortXMLElementType” maxOccurs=”unbounded”/>

</xsd:sequence>

<xsd:attribute name=”nunber” type=”xsd:int”/>

```
</xsd:complexType>
</xsd:element>
<xsd:element name="outputPorts">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="outputPort" type="PortXMLElementType" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="nunber" type="xsd:int"/>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="workflowBody">
<xsd:complexType>
<xsd:choice>
<xsd:sequence>
<xsd:element name="baseWorkflow" type="xsd:string"/>
<xsd:element name="unary-construct">
<xsd:complexType>
<xsd:sequence>
<xsd:choice minOccurs="0" maxOccurs="unbounded">
<xsd:element name="map" minOccurs="0" maxOccurs="unbounded">
<xsd:complexType>
<xsd:attribute name="mapPort" type="xsd:string"/>
</xsd:complexType>
</xsd:element>
```

<xsd:element name=”reduce” minOccurs=”0” maxOccurs=”unbounded”>

<xsd:complexType>

<xsd:attribute name=”basePort” type=”xsd:string”/>

<xsd:attribute name=”reducePort” type=”xsd:string”/>

</xsd:complexType>

</xsd:element>

<xsd:element name=”tree” minOccurs=”0” maxOccurs=”unbounded”>

<xsd:complexType>

<xsd:attribute name=”leftPort” type=”xsd:string”/>

<xsd:attribute name=”rightPort” type=”xsd:string”/>

</xsd:complexType>

</xsd:element>

<xsd:element name=”loop” minOccurs=”0” maxOccurs=”unbounded”>

<xsd:complexType>

<xsd:attribute name=”loopPort” type=”xsd:string”/>

<xsd:attribute name=”predicate” type=”xsd:string”/>

</xsd:complexType>

</xsd:element>

<xsd:element name=”conditional” minOccurs=”0” maxOccurs=”unbounded”>

<xsd:complexType>

<xsd:attribute name=”conditionalPort” type=”xsd:string”/>

<xsd:attribute name=”predicate” type=”xsd:string”/>

</xsd:complexType>

</xsd:element>

<xsd:element name=”curry” minOccurs=”0” maxOccurs=”unbounded”>

<xsd:complexType>

<xsd:sequence>

<xsd:element name=”inputMapping” type=”WorkflowPortMappingXMLElementType” minOccurs=”0” maxOccurs=”unbounded”/>

<xsd:element name=”assign” type=”WorkflowPortMappingXMLElementType” minOccurs=”0” maxOccurs=”unbounded”/>

<xsd:element name=”outputMapping” type=”WorkflowPortMappingXMLElementType” minOccurs=”0” maxOccurs=”unbounded”/>

</xsd:sequence>

<xsd:attribute name=”curryPort” type=”xsd:string”/>

<xsd:attribute name=”parameter” type=”xsd:string”/>

<xsd:attribute name=”parameterType” type=”xsd:string”/>

</xsd:complexType>

</xsd:element>

</xsd:choice>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

</xsd:sequence>

<xsd:sequence>

<xsd:element name=”taskComponent”>

<xsd:complexType>

<xsd:sequence>

<xsd:choice>

<xsd:sequence>

<xsd:element name=”wsdlURI” type=”xsd:string”/>

<xsd:element name=”serviceName” type=”xsd:string”/>

<xsd:element name=”operationName” type=”xsd:string”/>

</xsd:sequence>

<xsd:sequence>

<xsd:element name=”directory” type=”xsd:string”/>

<xsd:element name=”appName” type=”xsd:string”/>

</xsd:sequence>

<xsd:sequence>

<xsd:element name=”executable” type=”xsd:string”/>

<xsd:element name=”appName” type=”xsd:string”/>

</xsd:sequence>

</xsd:choice>

<xsd:element name=”taskInvocation”>

<xsd:complexType>

<xsd:sequence>

<xsd:element name=”operatingSystem”>

<xsd:simpleType>

<xsd:restriction base=”xsd:string”>

<xsd:enumeration value=”Windows”/>

<xsd:enumeration value=”Unix”/>

<xsd:enumeration value=”Linux”/>

<xsd:enumeration value=”Mac”/>

<xsd:enumeration value=”Unknown”/>

</xsd:restriction>

</xsd:simpleType>

</xsd:element>

<xsd:element name=”invocationMode”>

<xsd:simpleType>

<xsd:restriction base=”xsd:string”>

<xsd:enumeration value=”Local”/>

<xsd:enumeration value=”Remote”/>

</xsd:restriction>

</xsd:simpleType>

</xsd:element>

<xsd:element name=”interactionMode”>

<xsd:simpleType>

<xsd:restriction base=”xsd:string”>

<xsd:enumeration value=”Yes”/>

<xsd:enumeration value=”No”/>

</xsd:restriction>

</xsd:simpleType>

</xsd:element>

<xsd:element name=”invocationAuthentication” minOccurs=”0”>

<xsd:complexType>

<xsd:sequence>

<xsd:element name=”hostName” type=”xsd:string”/>

<xsd:element name=”userName” type=”xsd:string”/>

<xsd:element name=”password” type=”xsd:string”/>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

</xsd:sequence>

<xsd:attribute name=”taskType”>

<xsd:simpleType>

<xsd:restriction base=”xsd:string”>

<xsd:enumeration value=”WIndowsApplication”/>

<xsd:enumeration value=”WebService”/>

<xsd:enumeration value=”GridJob”/>

</xsd:restriction>

</xsd:simpleType>

</xsd:attribute>

</xsd:complexType>

</xsd:element>

<xsd:element name=”T2W”>

<xsd:complexType>

<xsd:sequence>

<xsd:element name=”inputs”>

<xsd:complexType>

<xsd:sequence>

<xsd:element name=”input” type=”TaskPortMappingXMLElementType” maxOccurs=”unbounded”/>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

<xsd:element name=”outputs”>

<xsd:complexType>

<xsd:sequence>

<xsd:element name=”output” type=”TaskPortMappingXMLElementType” maxOccurs=”unbounded”/>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

</xsd:sequence>

```
</xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:sequence>
<xsd:element name="workflowGraph">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="workflowInstances">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="workflowInstance" minOccurs="0" maxOccurs="unbounded">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="workflow" type="xsd:string"/>
</xsd:sequence>
<xsd:attribute name="id" type="xsd:string"/>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="dataChannels">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="dataChannel" type="DataChannelXMLElementType" minOccurs="0"
maxOccurs="unbounded"/>
</xsd:sequence>
```

```
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="G2W">
<xsd:complexType>
<xsd:sequence>
<xsd:element name="inputMapping" type="WorkflowPortMappingXMLElementType" minOc-
curs="0" maxOccurs="unbounded"/>
<xsd:element name="outputMapping" type="WorkflowPortMappingXMLElementType" minOc-
curs="0" maxOccurs="unbounded"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:sequence>
<xsd:element name="builtin" type="xsd:string"/>
</xsd:sequence>
</xsd:choice>
<xsd:attribute name="mode">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:enumeration value="unary-construct-based"/>
<xsd:enumeration value="primitive"/>
<xsd:enumeration value="graph-based"/>
<xsd:enumeration value="builtin"/>
```

```
</xsd:restriction>
</xsd:simpleType>
</xsd:attribute>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
<xsd:attribute name=”name” type=”xsd:string”/>
<xsd:attribute name=”root” type=”xsd:boolean”/>
</xsd:complexType>
<xsd:complexType name=”PortXMLElementType”>
<xsd:sequence>
<xsd:element name=”portID” type=”xsd:string”/>
<xsd:element name=”portName” type=”xsd:string”/>
<xsd:element name=”portType”>
<xsd:simpleType>
<xsd:restriction base=”xsd:string”>
<xsd:enumeration value=”String”/>
<xsd:enumeration value=”List”/>
<xsd:enumeration value=”Date”/>
<xsd:enumeration value=”Integer”/>
<xsd:enumeration value=”Double”/>
<xsd:enumeration value=”Decimal”/>
<xsd:enumeration value=”Boolean”/>
<xsd:enumeration value=”Uri”/>
<xsd:enumeration value=”File”/>
<xsd:enumeration value=”RelationBase”/>
<xsd:enumeration value=”CollectionBase”/>
```

<xsd:enumeration value=”Object”/>

</xsd:restriction>

</xsd:simpleType>

</xsd:element>

<xsd:element name=”portParameter” type=”xsd:string” minOccurs=”0”/>

<xsd:element name=”portDescription” type=”DescriptionXMLElementType” minOccurs=”0”/>

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name=”TaskPortMappingXMLElementType”>

<xsd:attribute name=”id” type=”xsd:string” use=”required”/>

<xsd:attribute name=”mode” type=”xsd:string” use=”required”/>

<xsd:attribute name=”name” type=”xsd:string” use=”required”/>

<xsd:attribute name=”type” type=”xsd:string” use=”required”/>

</xsd:complexType>

<xsd:complexType name=”DataChannelXMLElementType”>

<xsd:attribute name=”type”>

<xsd:simpleType>

<xsd:restriction base=”xsd:string”>

<xsd:enumeration value=”OneToOneDataChannel”/>

<xsd:enumeration value=”OneToManylDataChannel”/>

<xsd:enumeration value=”ManyToOnetDataChannel”/>

</xsd:restriction>

</xsd:simpleType>

</xsd:attribute>

<xsd:attribute name=”from” type=”xsd:string”/>

<xsd:attribute name=”to” type=”xsd:string”/>

</xsd:complexType>

<xsd:complexType name=”WorkflowPortMappingXMLElementType”>

<xsd:attribute name=”from” type=”xsd:string”/>

<xsd:attribute name=”to” type=”xsd:string”/>

</xsd:complexType>

<xsd:simpleType name=”DescriptionXMLElementType”>

<xsd:restriction base=”xsd:string”/>

</xsd:simpleType>

</xsd:schema>

APPENDIX B

Data Product Language (DPL)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
attributeFormDefault="unqualified" version="1.0.0">
<xsd:element name="dataProduct" type="DataProductXMLElementType"/>
<xsd:complexType name="DataProductXMLElementType">
<xsd:sequence>
<xsd:element name="description" type="xsd:string"/>
<xsd:element name="type">
<xsd:simpleType>
<xsd:restriction base="xsd:string">
<xsd:enumeration value="ScalarValue"/>
<xsd:enumeration value="File"/>
<xsd:enumeration value="List"/>
<xsd:enumeration value="Relation"/>
<xsd:enumeration value="Collection"/>
</xsd:restriction>
</xsd:simpleType>
</xsd:element>
<xsd:element name="data" type="DataXMLElementType"/>
</xsd:sequence>
<xsd:attribute name="name"/>
</xsd:complexType>
```

<xsd:complexType name=”DataXMLElementType”>

<xsd:choice>

<xsd:element name=”scalarValue” type=”ScalarValueXMLElementType”/>

<xsd:element name=”blob” type=”xsd:base64Binary”/>

<xsd:element name=”list” type=”ListXMLElementType”/>

<xsd:element name=”relation” type=”RelationXMLElementType”/>

<xsd:element name=”collection” type=”CollectionXMLElementType”/>

</xsd:choice>

</xsd:complexType>

<xsd:complexType name=”ScalarValueXMLElementType”>

<xsd:sequence>

<xsd:element name=”scalarType” type=”ScalarDataTypeEnumeration”/>

<xsd:element name=”value” type=”xsd:string”/>

</xsd:sequence>

</xsd:complexType>

<xsd:simpleType name=”ScalarDataTypeEnumeration”>

<xsd:restriction base=”xsd:string”>

<xsd:enumeration value=”String”/>

<xsd:enumeration value=”Date”/>

<xsd:enumeration value=”Integer”/>

<xsd:enumeration value=”Long”/>

<xsd:enumeration value=”Double”/>

<xsd:enumeration value=”Decimal”/>

<xsd:enumeration value=”Boolean”/>

<xsd:enumeration value=”Uri”/>

<xsd:enumeration value=”Blob”/>

</xsd:restriction>

```
</xsd:simpleType>
<xsd:complexType name="ListXMLElementType">
<xsd:sequence minOccurs="0" maxOccurs="unbounded">
<xsd:element name="dataProduct" type="DataProductXMLElementType"/>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="RelationXMLElementType">
<xsd:sequence>
<xsd:element name="schema">
<xsd:complexType>
<xsd:sequence maxOccurs="unbounded">
<xsd:element name="column" type="DataColumnXMLElementType"/>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
<xsd:element name="instance">
<xsd:complexType>
<xsd:sequence maxOccurs="unbounded">
<xsd:element name="row" type="DataRowXMLElementType"/>
</xsd:sequence>
<xsd:attribute name="count" type="xsd:integer"/>
</xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="DataColumnXMLElementType">
<xsd:sequence>
```

<xsd:element name=”columnName” type=”xsd:string”/>

<xsd:element name=”columnType” type=”ScalarDataTypeEnumeration”/>

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name=”DataRowXMLElementType”>

<xsd:sequence>

<xsd:element name=”dataElement” type=”xsd:string”/>

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name=”CollectionXMLElementType”>

<xsd:sequence>

<xsd:element name=”schema”>

<xsd:complexType>

<xsd:sequence maxOccurs=”unbounded”>

<xsd:element name=”key” type=”KeyXMLElementType”/>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

<xsd:element name=”nodeSchema”>

<xsd:complexType>

<xsd:sequence maxOccurs=”unbounded”>

<xsd:element name=”column” type=”DataColumnXMLElementType”/>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

<xsd:element name=”instance”>

<xsd:complexType>

<xsd:sequence maxOccurs=”unbounded”>

<xsd:element name=”pair” type=”PairXMLElementType”/>

</xsd:sequence>

<xsd:attribute name=”count” type=”xsd:integer”/>

</xsd:complexType>

</xsd:element>

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name=”KeyXMLElementType”>

<xsd:sequence>

<xsd:element name=”keyName” type=”xsd:string”/>

<xsd:element name=”keyType” type=”ScalarDataTypeEnumeration”/>

</xsd:sequence>

</xsd:complexType>

<xsd:complexType name=”PairXMLElementType”>

<xsd:sequence>

<xsd:element name=”key” type=”xsd:string”/>

<xsd:choice>

<xsd:element name=”relation” type=”RelationXMLElementType”/>

<xsd:element name=”collection” type=”CollectionXMLElementType”/>

</xsd:choice>

</xsd:sequence>

</xsd:complexType>

</xsd:schema>

APPENDIX C

WDSL Specification for Workflow Engine Web Services

<?xml version=”1.0” encoding=”utf-8”?>

<wsdl:definitions xmlns:soap=”http://schemas.xmlsoap.org/wsdl/soap/”

xmlns:tm=”http://microsoft.com/wsdl/mime/textMatching/”

xmlns:soapenc=”http://schemas.xmlsoap.org/soap/encoding/”

xmlns:mime=”http://schemas.xmlsoap.org/wsdl/mime/”

xmlns:tns=”http://dmsg2.cs.wayne.edu/view”

xmlns:s=”http://www.w3.org/2001/XMLSchema”

xmlns:soap12=”http://schemas.xmlsoap.org/wsdl/soap12/”

xmlns:http=”http://schemas.xmlsoap.org/wsdl/http/”

targetNamespace=”http://dmsg2.cs.wayne.edu/view”

xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”>

<wsdl:types>

<s:schema elementFormDefault=”qualified” targetNamespace=”http://dmsg2.cs.wayne.edu/view”>

<s:element name=”GetWorkflowList”>

<s:complexType>

<s:sequence>

<s:element minOccurs=”0” maxOccurs=”1” name=”type” type=”s:string” />

</s:sequence>

</s:complexType>

</s:element>

<s:element name=”GetWorkflowListResponse”>

<s:complexType>

<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="GetWorkflowListResult"
type="tns:ArrayOfWorkflowInfo" />
</s:sequence>
</s:complexType>
</s:element>
<s:complexType name="ArrayOfWorkflowInfo">
<s:sequence>
<s:element minOccurs="0" maxOccurs="unbounded" name="WorkflowInfo" nillable="true"
type="tns:WorkflowInfo" />
</s:sequence>
</s:complexType>
<s:complexType name="WorkflowInfo">
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="workflowID" type="s:string" />
<s:element minOccurs="0" maxOccurs="1" name="workflowName" type="s:string" />
<s:element minOccurs="0" maxOccurs="1" name="workflowDescription" type="s:string"
/>
<s:element minOccurs="0" maxOccurs="1" name="workflowType" type="s:string" />
</s:sequence>
</s:complexType>
<s:element name="GetWorkflow">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="workflowID" type="s:string" />
</s:sequence>
</s:complexType>

```
</s:element>
<s:element name=”GetWorkflowResponse”>
<s:complexType>
<s:sequence>
<s:element minOccurs=”0” maxOccurs=”1” name=”GetWorkflowResult” type=”s:base64Binary”
/>
</s:sequence>
</s:complexType>
</s:element>
<s:element name=”RegisterWorkflow”>
<s:complexType>
<s:sequence>
<s:element minOccurs=”0” maxOccurs=”1” name=”workflowSpecification” type=”s:base64Binary”
/>
<s:element minOccurs=”0” maxOccurs=”1” name=”visualizationFile” type=”s:base64Binary”
/>
</s:sequence>
</s:complexType>
</s:element>
<s:element name=”RegisterWorkflowResponse”>
<s:complexType>
<s:sequence>
<s:element minOccurs=”1” maxOccurs=”1” name=”RegisterWorkflowResult” type=”s:int”
/>
</s:sequence>
</s:complexType>
</s:element>
```

<s:element name=”DeleteWorkflow”>

<s:complexType>

<s:sequence>

<s:element minOccurs=”0” maxOccurs=”1” name=”workflowID” type=”s:string” />

</s:sequence>

</s:complexType>

</s:element>

<s:element name=”DeleteWorkflowResponse”>

<s:complexType>

<s:sequence>

<s:element minOccurs=”1” maxOccurs=”1” name=”DeleteWorkflowResult” type=”s:int” />

</s:sequence>

</s:complexType>

</s:element>

<s:element name=”ExecuteWorkflow”>

<s:complexType>

<s:sequence>

<s:element minOccurs=”1” maxOccurs=”1” name=”workflowID” type=”s:int” />

<s:element minOccurs=”0” maxOccurs=”1” name=”inputData”

type=”tns:ArrayOfDataflowTransformation” />

</s:sequence>

</s:complexType>

</s:element>

<s:complexType name=”ArrayOfDataflowTransformation”>

<s:sequence>

<s:element minOccurs=”0” maxOccurs=”unbounded” name=”DataflowTransformation” nil-

lable=”true” type=”tns:DataflowTransformation” />

```
</s:sequence>
</s:complexType>
<s:complexType name="DataflowTransformation">
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="portID" type="s:string" />
<s:element minOccurs="0" maxOccurs="1" name="dataType" type="s:string" />
<s:element minOccurs="0" maxOccurs="1" name="dataID" type="s:string" />
</s:sequence>
</s:complexType>
<s:element name="ExecuteWorkflowResponse">
<s:complexType>
<s:sequence>
<s:element minOccurs="0" maxOccurs="1" name="ExecuteWorkflowResult"
type="tns:ArrayOfDataflowTransformation" />
</s:sequence>
</s:complexType>
</s:element>
</s:schema>
</wsdl:types>
<wsdl:message name="GetWorkflowListSoapIn">
<wsdl:part name="parameters" element="tns:GetWorkflowList" />
</wsdl:message>
<wsdl:message name="GetWorkflowListSoapOut">
<wsdl:part name="parameters" element="tns:GetWorkflowListResponse" />
</wsdl:message>
<wsdl:message name="GetWorkflowSoapIn">
<wsdl:part name="parameters" element="tns:GetWorkflow" />
```

</wsdl:message>

<wsdl:message name=”GetWorkflowSoapOut”>

<wsdl:part name=”parameters” element=”tns:GetWorkflowResponse” />

</wsdl:message>

<wsdl:message name=”RegisterWorkflowSoapIn”>

<wsdl:part name=”parameters” element=”tns:RegisterWorkflow” />

</wsdl:message>

<wsdl:message name=”RegisterWorkflowSoapOut”>

<wsdl:part name=”parameters” element=”tns:RegisterWorkflowResponse” />

</wsdl:message>

<wsdl:message name=”DeleteWorkflowSoapIn”>

<wsdl:part name=”parameters” element=”tns:DeleteWorkflow” />

</wsdl:message>

<wsdl:message name=”DeleteWorkflowSoapOut”>

<wsdl:part name=”parameters” element=”tns:DeleteWorkflowResponse” />

</wsdl:message>

<wsdl:message name=”ExecuteWorkflowSoapIn”>

<wsdl:part name=”parameters” element=”tns:ExecuteWorkflow” />

</wsdl:message>

<wsdl:message name=”ExecuteWorkflowSoapOut”>

<wsdl:part name=”parameters” element=”tns:ExecuteWorkflowResponse” />

</wsdl:message>

<wsdl:portType name=”WorkflowEngineSoap”>

<wsdl:operation name=”GetWorkflowList”>

<wsdl:documentation xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”>

Get information of a category of workflows by workflow type:

”GraphBased”,”TaskBased”,”UConstructBased”, </wsdl:documentation>

<wsdl:input message=”tns:GetWorkflowListSoapIn” />

<wsdl:output message=”tns:GetWorkflowListSoapOut” />

</wsdl:operation>

<wsdl:operation name=”GetWorkflow”>

<wsdl:documentation xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”>

Get workflow specification by workflow id</wsdl:documentation>

<wsdl:input message=”tns:GetWorkflowSoapIn” />

<wsdl:output message=”tns:GetWorkflowSoapOut” />

</wsdl:operation>

<wsdl:operation name=”RegisterWorkflow”>

<wsdl:documentation xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”>

Register workflow with workflow definition file in SWL and workflow visualizationFile if necessary for workbench</wsdl:documentation>

<wsdl:input message=”tns:RegisterWorkflowSoapIn” />

<wsdl:output message=”tns:RegisterWorkflowSoapOut” />

</wsdl:operation>

<wsdl:operation name=”DeleteWorkflow”>

<wsdl:documentation xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”>

Delete a workflow by workflow id</wsdl:documentation>

<wsdl:input message=”tns:DeleteWorkflowSoapIn” />

<wsdl:output message=”tns:DeleteWorkflowSoapOut” />

</wsdl:operation>

<wsdl:operation name=”ExecuteWorkflow”>

<wsdl:documentation xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”>

Run a Workflow with inputData</wsdl:documentation>

<wsdl:input message=”tns:ExecuteWorkflowSoapIn” />

<wsdl:output message=”tns:ExecuteWorkflowSoapOut” />

```
</wsdl:operation>
</wsdl:portType>
<wsdl:binding name="WorkflowEngineSoap" type="tns:WorkflowEngineSoap">
<soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
<wsdl:operation name="GetWorkflowList">
<soap:operation soapAction="http://dmsg2.cs.wayne.edu/view/GetWorkflowList" style="document"
/>
<wsdl:input>
<soap:body use="literal" />
</wsdl:input>
<wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="GetWorkflow">
<soap:operation soapAction="http://dmsg2.cs.wayne.edu/view/GetWorkflow" style="document"
/>
<wsdl:input>
<soap:body use="literal" />
</wsdl:input>
<wsdl:output>
<soap:body use="literal" />
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="RegisterWorkflow">
<soap:operation soapAction="http://dmsg2.cs.wayne.edu/view/RegisterWorkflow" style="document"
/>
```

<wsdl:input>

<soap:body use=”literal” />

</wsdl:input>

<wsdl:output>

<soap:body use=”literal” />

</wsdl:output>

</wsdl:operation>

<wsdl:operation name=”DeleteWorkflow”>

<soap:operation soapAction=”http://dmsg2.cs.wayne.edu/view/DeleteWorkflow” style=”document”

/>

<wsdl:input>

<soap:body use=”literal” />

</wsdl:input>

<wsdl:output>

<soap:body use=”literal” />

</wsdl:output>

</wsdl:operation>

<wsdl:operation name=”ExecuteWorkflow”>

<soap:operation soapAction=”http://dmsg2.cs.wayne.edu/view/ExecuteWorkflow” style=”document”

/>

<wsdl:input>

<soap:body use=”literal” />

</wsdl:input>

<wsdl:output>

<soap:body use=”literal” />

</wsdl:output>

</wsdl:operation>

```
</wsdl:binding>
<wsdl:binding name="WorkflowEngineSoap12" type="tns:WorkflowEngineSoap">
<soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
<wsdl:operation name="GetWorkflowList">
<soap12:operation soapAction="http://dmsg2.cs.wayne.edu/view/GetWorkflowList" style="document"
/>
<wsdl:input>
<soap12:body use="literal" />
</wsdl:input>
<wsdl:output>
<soap12:body use="literal" />
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="GetWorkflow">
<soap12:operation soapAction="http://dmsg2.cs.wayne.edu/view/GetWorkflow" style="document"
/>
<wsdl:input>
<soap12:body use="literal" />
</wsdl:input>
<wsdl:output>
<soap12:body use="literal" />
</wsdl:output>
</wsdl:operation>
<wsdl:operation name="RegisterWorkflow">
<soap12:operation soapAction="http://dmsg2.cs.wayne.edu/view/RegisterWorkflow" style="document"
/>
<wsdl:input>
```

<soap12:body use=”literal” />

</wsdl:input>

<wsdl:output>

<soap12:body use=”literal” />

</wsdl:output>

</wsdl:operation>

<wsdl:operation name=”DeleteWorkflow”>

<soap12:operation soapAction=”http://dmsg2.cs.wayne.edu/view/DeleteWorkflow” style=”document”

/>

<wsdl:input>

<soap12:body use=”literal” />

</wsdl:input>

<wsdl:output>

<soap12:body use=”literal” />

</wsdl:output>

</wsdl:operation>

<wsdl:operation name=”ExecuteWorkflow”>

<soap12:operation soapAction=”http://dmsg2.cs.wayne.edu/view/ExecuteWorkflow” style=”document”

/>

<wsdl:input>

<soap12:body use=”literal” />

</wsdl:input>

<wsdl:output>

<soap12:body use=”literal” />

</wsdl:output>

</wsdl:operation>

</wsdl:binding>

<wsdl:service name=”WorkflowEngine”>

<wsdl:port name=”WorkflowEngineSoap” binding=”tns:WorkflowEngineSoap”>

<soap:address location=”http://localhost:5977/WorkflowEngine.asmx” />

</wsdl:port>

<wsdl:port name=”WorkflowEngineSoap12” binding=”tns:WorkflowEngineSoap12”>

<soap12:address location=”http://localhost:5977/WorkflowEngine.asmx” />

</wsdl:port>

</wsdl:service>

</wsdl:definitions>

APPENDIX D

WDSL Specification for Data Product Manager Web Services

<?xml version=”1.0” encoding=”utf-8”?>

<wsdl:definitions xmlns:soap=”http://schemas.xmlsoap.org/wsdl/soap/”

xmlns:tm=”http://microsoft.com/wsdl/mime/textMatching/”

xmlns:soapenc=”http://schemas.xmlsoap.org/soap/encoding/”

xmlns:mime=”http://schemas.xmlsoap.org/wsdl/mime/”

xmlns:tns=”http://dmsg2.cs.wayne.edu/view”

xmlns:s=”http://www.w3.org/2001/XMLSchema”

xmlns:soap12=”http://schemas.xmlsoap.org/wsdl/soap12/”

xmlns:http=”http://schemas.xmlsoap.org/wsdl/http/”

targetNamespace=”http://dmsg2.cs.wayne.edu/view”

xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”>

<wsdl:types>

<s:schema elementFormDefault=”qualified” targetNamespace=”http://dmsg2.cs.wayne.edu/view”>

<s:element name=”GetDataProductInfoList”>

<s:complexType>

<s:sequence>

<s:element minOccurs=”0” maxOccurs=”1” name=”dataProductType” type=”s:string” />

</s:sequence>

</s:complexType>

</s:element>

<s:element name=”GetDataProductInfoListResponse”>

<s:complexType>

<s:sequence>

<s:element minOccurs="0" maxOccurs="1" name="GetDataProductInfoListResult"

type="tns:ArrayOfDataProductInfo" />

</s:sequence>

</s:complexType>

</s:element>

<s:complexType name="ArrayOfDataProductInfo">

<s:sequence>

<s:element minOccurs="0" maxOccurs="unbounded" name="DataProductInfo"

type="tns:DataProductInfo" />

</s:sequence>

</s:complexType>

<s:complexType name="DataProductInfo">

<s:sequence>

<s:element minOccurs="0" maxOccurs="1" name="dataID" type="s:string" />

<s:element minOccurs="0" maxOccurs="1" name="dataName" type="s:string" />

<s:element minOccurs="0" maxOccurs="1" name="dataDescription" type="s:string" />

<s:element minOccurs="0" maxOccurs="1" name="dataType" type="s:string" />

</s:sequence>

</s:complexType>

<s:element name="DeleteDataProduct">

<s:complexType>

<s:sequence>

<s:element minOccurs="0" maxOccurs="1" name="dataID" type="s:string" />

</s:sequence>

</s:complexType>

</s:element>

<s:element name=”DeleteDataProductResponse”>

<s:complexType />

</s:element>

<s:element name=”RegisterDataProduct”>

<s:complexType>

<s:sequence>

<s:element minOccurs=”0” maxOccurs=”1” name=”dataProductDescription” type=”s:base64Binary”

/>

</s:sequence>

</s:complexType>

</s:element>

<s:element name=”RegisterDataProductResponse”>

<s:complexType>

<s:sequence>

<s:element minOccurs=”0” maxOccurs=”1” name=”RegisterDataProductResult” type=”s:string”

/>

</s:sequence>

</s:complexType>

</s:element>

<s:element name=”GetDataProduct”>

<s:complexType>

<s:sequence>

<s:element minOccurs=”0” maxOccurs=”1” name=”dataID” type=”s:string” />

</s:sequence>

</s:complexType>

</s:element>

<s:element name=”GetDataProductResponse”>

<s:complexType>

<s:sequence>

<s:element minOccurs="0" maxOccurs="1" name="GetDataProductResult" type="s:base64Binary"

/>

</s:sequence>

</s:complexType>

</s:element>

</s:schema>

</wsdl:types>

<wsdl:message name="GetDataProductInfoListSoapIn">

<wsdl:part name="parameters" element="tns:GetDataProductInfoList" />

</wsdl:message>

<wsdl:message name="GetDataProductInfoListSoapOut">

<wsdl:part name="parameters" element="tns:GetDataProductInfoListResponse" />

</wsdl:message>

<wsdl:message name="DeleteDataProductSoapIn">

<wsdl:part name="parameters" element="tns:DeleteDataProduct" />

</wsdl:message>

<wsdl:message name="DeleteDataProductSoapOut">

<wsdl:part name="parameters" element="tns:DeleteDataProductResponse" />

</wsdl:message>

<wsdl:message name="RegisterDataProductSoapIn">

<wsdl:part name="parameters" element="tns:RegisterDataProduct" />

</wsdl:message>

<wsdl:message name="RegisterDataProductSoapOut">

<wsdl:part name="parameters" element="tns:RegisterDataProductResponse" />

</wsdl:message>

<wsdl:message name="GetDataProductSoapIn">

<wsdl:part name="parameters" element="tns:GetDataProduct" />

</wsdl:message>

<wsdl:message name="GetDataProductSoapOut">

<wsdl:part name="parameters" element="tns:GetDataProductResponse" />

</wsdl:message>

<wsdl:portType name="DataProductManagementSoap">

<wsdl:operation name="GetDataProductList">

<wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

Return Data Product List By Data Product Type: Relation, Collection, List</wsdl:documentation>

<wsdl:input name="GetDataProductInfoList" message="tns:GetDataProductInfoListSoapIn"

/>

<wsdl:output name="GetDataProductInfoList" message="tns:GetDataProductInfoListSoapOut"

/>

</wsdl:operation>

<wsdl:operation name="DeleteDataProduct">

<wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

DELETE A Data Product By Data ID.</wsdl:documentation>

<wsdl:input message="tns:DeleteDataProductSoapIn" />

<wsdl:output message="tns:DeleteDataProductSoapOut" />

</wsdl:operation>

<wsdl:operation name="RegisterDataProduct">

<wsdl:documentation xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">

Register A Data Product by Data Product Description File .</wsdl:documentation>

<wsdl:input message="tns:RegisterDataProductSoapIn" />

<wsdl:output message="tns:RegisterDataProductSoapOut" />

</wsdl:operation>

<wsdl:operation name=”GetDataProductDescription”>

<wsdl:documentation xmlns:wsdl=”http://schemas.xmlsoap.org/wsdl/”>

Get A Data Product Description File By Data ID.</wsdl:documentation>

<wsdl:input name=”GetDataProduct” message=”tns:GetDataProductSoapIn” />

<wsdl:output name=”GetDataProduct” message=”tns:GetDataProductSoapOut” />

</wsdl:operation>

</wsdl:portType>

<wsdl:binding name=”DataProductManagementSoap” type=”tns:DataProductManagementSoap”>

<soap:binding transport=”http://schemas.xmlsoap.org/soap/http” />

<wsdl:operation name=”GetDataProductList”>

<soap:operation soapAction=”http://dmsg2.cs.wayne.edu/view/GetDataProductInfoList”

style=”document” />

<wsdl:input name=”GetDataProductInfoList”>

<soap:body use=”literal” />

</wsdl:input>

<wsdl:output name=”GetDataProductInfoList”>

<soap:body use=”literal” />

</wsdl:output>

</wsdl:operation>

<wsdl:operation name=”DeleteDataProduct”>

<soap:operation soapAction=”http://dmsg2.cs.wayne.edu/view/DeleteDataProduct”

style=”document” />

<wsdl:input>

<soap:body use=”literal” />

</wsdl:input>

<wsdl:output>

<soap:body use=”literal” />

</wsdl:output>

</wsdl:operation>

<wsdl:operation name=”RegisterDataProduct”>

<soap:operation soapAction=”http://dmsg2.cs.wayne.edu/view/RegisterDataProduct”

style=”document” />

<wsdl:input>

<soap:body use=”literal” />

</wsdl:input>

<wsdl:output>

<soap:body use=”literal” />

</wsdl:output>

</wsdl:operation>

<wsdl:operation name=”GetDataProductDescription”>

<soap:operation soapAction=”http://dmsg2.cs.wayne.edu/view/GetDataProduct”

style=”document” />

<wsdl:input name=”GetDataProduct”>

<soap:body use=”literal” />

</wsdl:input>

<wsdl:output name=”GetDataProduct”>

<soap:body use=”literal” />

</wsdl:output>

</wsdl:operation>

</wsdl:binding>

<wsdl:binding name=”DataProductManagementSoap12” type=”tns:DataProductManagementSoap”>

<soap12:binding transport=”http://schemas.xmlsoap.org/soap/http” />

<wsdl:operation name=”GetDataProductList”>

<soap12:operation soapAction=”http://dmsg2.cs.wayne.edu/view/GetDataProductInfoList”

style=”document” />

<wsdl:input name=”GetDataProductInfoList”>

<soap12:body use=”literal” />

</wsdl:input>

<wsdl:output name=”GetDataProductInfoList”>

<soap12:body use=”literal” />

</wsdl:output>

</wsdl:operation>

<wsdl:operation name=”DeleteDataProduct”>

<soap12:operation soapAction=”http://dmsg2.cs.wayne.edu/view/DeleteDataProduct”

style=”document” />

<wsdl:input>

<soap12:body use=”literal” />

</wsdl:input>

<wsdl:output>

<soap12:body use=”literal” />

</wsdl:output>

</wsdl:operation>

<wsdl:operation name=”RegisterDataProduct”>

<soap12:operation soapAction=”http://dmsg2.cs.wayne.edu/view/RegisterDataProduct”

style=”document” />

<wsdl:input>

<soap12:body use=”literal” />

</wsdl:input>

<wsdl:output>

<soap12:body use=”literal” />

</wsdl:output>

```
</wsdl:operation>
<wsdl:operation name="GetDataProductDescription">
<soap12:operation soapAction="http://dmsg2.cs.wayne.edu/view/GetDataProduct"
style="document" />
<wsdl:input name="GetDataProduct">
<soap12:body use="literal" />
</wsdl:input>
<wsdl:output name="GetDataProduct">
<soap12:body use="literal" />
</wsdl:output>
</wsdl:operation>
</wsdl:binding>
<wsdl:service name="DataProductManagement">
<wsdl:port name="DataProductManagementSoap" binding="tns:DataProductManagementSoap">
<soap:address location="http://localhost:6077/DataProductManagement.asmx" />
</wsdl:port>
<wsdl:port name="DataProductManagementSoap12" binding="tns:DataProductManagementSoap12">
<soap12:address location="http://localhost:6077/DataProductManagement.asmx" />
</wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

BIBLIOGRAPHY

[1] Amazon Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/.

[2] Climate simulation at sandia national laboratories. http://www.cs.sandia.gov/capabilities/ClimateSimulation.

[3] DAX schema. http://pegasus.isi.edu/docs/schemas/dax-2.0/dax-2.0.html.

[4] FITS. http://fits.gsfc.nasa.gov/.

[5] HDF5. http://hdf.ncsa.uiuc.edu/HDF5/.

[6] Introduction to Amazon Elastic MapReduce. http://awsmedia.s3.amazonaws.com/pdf/introduction-to-amazon-elastic-mapreduce.pdf.

[7] ISO/IEC 9075-1:2008 Framework (SQL/Framework). http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45498.

[8] NetCDF. http://www.unidata.ucar.edu/software/netcdf/.

[9] The supercomputing science consortium. http://www.sc-2.psc.edu/.

[10] Top 500 list of world's supercomputers. http://www.top500.org/.

[11] W3C Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[12] Web services business process execution language version 2.0. http://www.oasis-open.org/committees/wsbpel/.

[13] XQuery 1.0 and XPath 2.0 Data Model (XDM) (Second Edition). http://www.w3.org/TR/xpath-datamodel/.

[14] Workflow management coalition. terminology and glossary. Technical report, Workflow Management Coalition, Brussels, 1996. http://www.aiim.org/wfmc/.

[15] Xscufl language reference, 2004. http://www.ebi.ac.uk/~tmo/mygrid/XScuflSpecification.html.

[16] Process definition interface – XML process definition language, (WFMCTC-1025). Technical report, Workflow Management Coalition, 2005.

[17] W. Aalst, A. van der, B. Hofstede, and A. Kiepuszewski. Advanced workflow patterns. In O. Etzion en P. Scheuermann, editor, *International Conference on Cooperative Information Systems*, volume 1901, pages 18–29, 2000.

[18] M. Adams, A. ter Hofstede, D. Edmond, and W. van der Aalst. Facilitating flexibility and dynamic exception handling in workflows through worklets. In *International Conference on Advanced Information Systems Engineering*, pages 45–50, 2005.

[19] A. Akram, D. Meredith, and R. Allan. Evaluation of BPEL to scientific workflows. In *IEEE International Symposium on Cluster Computing and the Grid*, volume 1, pages 269 – 274, 2006.

[20] G. Allen, K. Davis, K. Dolkas, N. Doulamis, T. Goodale, T. Kielmann, A. Merzky, J. Nabrzyski, J. Pukacki, T. Radke, M. Russell, E. Seidel, 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*, 17:449–466, 2003.

[21] G. Alonso, B. Reinwald, and C. Mohan. WIDE - a distributed architecture for workflow management. In *International Workshop on Research Issues in Data Engineering*, pages 76 – 79, 1997.

[22] M. Alt, A. Hoheisel, H. Pohl, and S. Gorlatch. A Grid workflow language using high-level Petri Nets. In *International Conference on Parallel Processing and Applied Mathematics*, pages 715–722, 2005.

[23] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business process execution language for Web services, version 1.1. 2003.

[24] A. Barker and J. Hemert. Scientific workflow: A survey and research directions. In *The Third Grid Applications and Middleware Workshop*, 2007.

[25] G. Bell, T. Hey, and A. Szalay. Beyond the data deluge. *Science*, 323(5919):1297–1298, 2009.

[26] S. Bowers and B. Ludäscher. Actor-oriented design of scientific workflows. In *International Conference on Conceptual Modeling*, pages 369–384, 2005.

[27] S. Bowers, B. Ludäscher, A. Ngu, and T. Critchlow. Enabling scientific workflow reuse through structured composition of dataflow and control-flow. *International Conference on Data Engineering Workshops*, 0:70–80, 2006.

[28] M. Boylan-Kolchin. Resolving cosmic structure formation with the Millennium-II simulation. *Monthly Notices of the Royal Astronomical Society*, 398(3):1150–1160, 2009.

[29] J. Brown, C. Ferner, T. Hudson, A. Stapleton, R. Vettera, T. Carland, A. Martin, J. Martin, A. Rawls, W. Shipman, and M. Wood. GridNexus: A Grid services scientific workflow system. *International Journal of Computer Information Science*, 6(2):72–82, 2005.

[30] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal in Computer Simulation*, 4:155–182, 1994.

[31] L. Cabellos, I. Plasencia, E. Fernández-del Castillo, M. Owsiak, B. Palak, and Plóciennik M. Scientific workflow orchestration interoperating HTC and HPC resources. *Computer Physics Communications*, 182(4):890–897, 2011.

[32] S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. Vo. VisTrails: visualization meets data management. In *ACM SIGMOD International Conference on Management of Data*, pages 745–747, 2006.

[33] A. Charfi and M. Mezini. Aspect-oriented workflow languages. *OnTheMove Federated Conferences*, 1:183–200, 2006.

[34] A. Chervenak, R. Schuler, C. Kesselman, S. Koranda, and B. Moe. Wide area data replication for scientific collaborations. In *IEEE/ACM International Workshop on Grid Computing*, pages 1–8, 2005.

[35] D. Churches, G. Gombas, A. Harrison, J. Maassen, C. Robinson, M. Shields, I. Taylor, and I. Wang. Programming scientific and distributed workflow with Triana services. *Concurrent Computing : Practice and Experience (Special Issue: Workflow in Grid Systems)*, 18(10):1021–1037, 2006.

[36] H. Claus and A. Gustavo. Exception handling in workflow management systems. *IEEE Transactions on Software Engineering*, 26(10):943–958, 2000.

[37] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[38] E. F. Codd. Relational completeness of data base sublanguages. In *Database Systems*, pages 65–98, 1972.

[39] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4):397–434, 1979.

[40] P. Cox, R. Betts, C. Jones, S. Spall, and I. Totterdall. Acceleration of global warming due to carbon-cycle feedbacks in a coupled climate model. *Nature*, 408:184–187, 2000.

[41] V. Curcin, M. Ghanem, Y. Guo, M. Kohler, A. Rowe, J. Syed, and Wendel P. Discovery Net: Towards a Grid of knowledge discovery. In *ACM International Conference on Knowledge Discovery and Data Mining.*, pages 23–26, 2002.

[42] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *USENIX Symposium on Operating Systems*, pages 137–150, 2004.

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

[44] E. Deelman and Y. Gil. Workshop on the challenges of scientific workflows. Technical report, Information Sciences Institute, University of Southern California, 2006.

[45] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, B. Berriman, J. Good, A. Laity, J. Jacob, and D. Katz. Pegasus: a framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[46] F. DeRemer and H. Kron. Programming-in-the-Large versus Programming-in-the-Small. In *Fachtagung über Programmiersprachen*, pages 80–89.

[47] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *International Conference on Database Theory*, pages 252–267, 2009.

[48] L. Dou, D. Zinn, T. McPhillips, S. Köhler, S. Riddle, S. Bowers, and B. Ludäscher. Scientific workflow design 2.0: Demonstrating streaming data collections in Kepler. In *IEEE International Conference on Data Engineering*, pages 1296–1299, 2011.

[49] T. Fahringer, R. Prodan, R. Duan, J. Hofer, F. Nadeem, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H.-L. Truong, A. Villazon, and M. Wieczorek. ASKALON: A development and grid computing environment for scientific workflows. *Workflows for eScience, Scientific Workflows for Grids*.

[50] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with AGWL: an abstract grid workflow language. *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 676–685, 2005.

[51] T. Fahringer, J. Qin, and S. Hainzer. Specification of grid workflow applications with AGWL: an abstract grid workflow language. *IEEE International Symposium on Cluster Computing and the Grid*, 2:676–685, 2005.

[52] X. Fei, S. Lu, T. Breithaupt, J. Hardege, and J. Ram. Modeling matefinding behavior of the swarming polychaete, Nereis Succinea, with TangoInSilico, a scientific workflow based simulation system for sexual searching. *Invertebrate Reproduction and Development*, 52(1-2):69–80, 2008.

[53] X. Fei, S. Lu, and C. Lin. A MapReduce-enabled scientific workflow composition framework. In *IEEE International Conference on Web Services*, pages 663–670, 2009.

[54] Ludwig Institute for Cancer Research. New computational tool for cancer treatment. `http://www.sciencedaily.com/releases/2010/01/100129151756.htm`.

[55] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: a virtual data system for representing, querying, and automating data derivation. In *Scientific and Statistical Database Management Conference*, pages 37–46, 2002.

[56] J. Freire, C. Silva, S. Callahan, E. Santos, C. Scheidegger, and H. Vo. Managing rapidly-evolving scientific workflows. In *International Provenance and Annotation Workshop*, pages 10–18, 2006.

[57] C. Fritz, R. Hull, and J. Su. Automatic construction of simple artifact-based business processes. In *International Conference on Database Theory*, pages 225–238, 2009.

[58] O. Gelly. LAU software system: A high-level data-driven language for parallel processing. In *International Conference on Parallel Processing*, page 255, 1976.

[59] T. Glatard, J. Montagnat, D. Lingrand, and X. Pennec. Flexible and efficient workflow deployment of data-intensive applications on Grids with MOTEUR. *International Journal of High Performance Computing Applications*, 22(3):347–360, 2008.

[60] A. Goderis, C. Brooks, I. Altintas, E. Lee, and G. Carol. Composing different models of computation in Kepler and Ptolemy II. In *International Conference on Computational Science*, pages 182–190, 2007.

[61] D. Goodman. Introduction and evaluation of Martlet: a scientific workflow language for abstracted parallelisation. In *WWW*, pages 983–992, 2007.

[62] J. Gray, D. Liu, M. Nieto-Santisteban, A. Szalay, D. DeWitt, and G. Heber. Scientific data management in the coming decade. *ACM SIGMOD Record*, 34(4):34–41, 2005.

[63] S. Gregory and M. Paschali. A Prolog-based language for workflow programming. *COORDINATION*, pages 56–75, 2007.

[64] J. Han, Y. Cho, and J. Choi. A workflow language based on structural context model for ubiquitous computing. *IEEE International Conference on Embedded and Ubiquitous Computing*, pages 879–889, 2005.

[65] D. Handl. HotFlow - a visual language for workflow applications in e-commerce. *Visual Languages*, pages 185–186, 1999.

[66] P. Heinl and H. Schuster. Towards a highly scalable architecture of workflow management systems. In *International Conference on Database and Expert System Applications*, 1996.

[67] T. Hey, S. Tansley, and K. Tolle. *The Fourth Paradigm: Data-Intensive Scientific Discovery*. Microsoft Research, 2009.

[68] D. Hollingsworth. The workflow reference model. *The Workflow Management Coalition*, 1994.

[69] Y. Huang and Q. Huang. Extensions to Web service techniques for integrating Jini into a service-oriented architecture for the grid. In *International Conference on Computational Science*, pages 254–263, 2003.

[70] Y. Huang and Q. Huang. WS-based workflow description language for message passing. In *IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 558–565, 2005.

[71] Y. Huang and Shan M. Policies in a resource manager of workflow systems: Modeling, enforcement and management. Technical Report HPL-98-156, HP Software Technology Laboratory, 1998.

[72] C. Hunt, C. Ferner, and J. Brown. JXPL: an XML-based scripting language for workflow execution in a grid environment. In *IEEE SoutheastCon*, pages 345–350, 2005.

[73] W. Johnston, Hanna J., and R. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[74] W. Johnston, Hanna J., and R. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 2004.

[75] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, Y. Lafon, C. Barreto, and G. Brown. Web services choreography description language version 1.0. 2005.

[76] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, C. von Riegen, P. Schmidt, and I. Trickovic. *WS-BPEL Extension for Sub-processes - BPEL-SPE*. Joint white paper, IBM and SAP, 2005.

[77] M. Kloppmann, D. Koenig, F. Leymann, G. Pfau, A. Rickayzen, and Schmidt P. Trickovic I. von Riegen, C. *WS-BPEL Extension for People C BPEL4People*. Joint white paper, IBM and SAP, 2005.

[78] C. Lee, S. Matsuoka, D. Talia, A. Sussmann, M. Uller, G. Allen, and J. Saltz. A Grid programming primer. Technical report, Global Grid Forum, 2001.

[79] E. Lee and S. Neuendorffer. MoML a modeling markup language in XML version 0.4. Technical report, University of California at Berkeley, March 2000.

[80] F. Leymann. Web services flow language (WSFL 1.0). Technical report, IBM Software Group, 2001.

[81] F. Leymann and W. Altenhuber. Managing business processes as an information resource. *IBM Systems Journal*, 33(2):36–47, 1994.

[82] C. Lim, S. Lu, A. Chebotko, and F. Fotouhi. Prospective and retrospective provenance collection in scientific workflow environments. In *International Conference on Services Computing*, pages 449–456, 2010.

[83] C. Lim, S. Lu, A. Chebotko, and F. Fotouhi. OPQL: A first OPM-Level query language for scientific workflow provenance. In *International Conference on Services Computing*, pages 136–143, 2011.

[84] C. Lim, S. Lu, A. Chebotko, and F. Fotouhi. Storing, reasoning, and querying OPM-Compliant scientific workflow provenance using relational databases. *Future Generation Computer Systems*, 27(6):781–789, 2011.

[85] 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*, 2(1):79–92, 2009.

[86] C. Lin, S. Lu, X. Fei, D. Pai, and J. Hua. A task abstraction and mapping approach to the shimming problem in scientific workflows. In *IEEE International Conference on Services Computing*, pages 284–291, 2009.

[87] D. Liu and M. Franklin. GridDB: a data-centric overlay for scientific grids. In *International Conference on Very large data bases*, pages 600–611. VLDB Endowment, 2004.

[88] S. Lu and J. Zhang. Collaborative scientific workflows. In *IEEE International Conference on Web Services*, pages 527–534, 2009.

[89] 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. *Concurrent Computing : Practice and Experience*, 18(10):1039–1065, 2006.

[90] D. Martin, M. Burstein, J. Hobbs, O. Lassila, D. McDermott, S. McIlraith, S. Narayanan, M. Paolucci, B. Parsia, T. Payne, E. Sirin, N. Srinivasan, and K. Sycara. OWL-S: Semantic markup for Web services. Retrieved from `http://www.w3.org/Submission/OWL-S/`.

[91] T. McPhillips, S. Bowers, and B. Ludäscher. Collection-oriented scientific workflows for integrating and analyzing biological data. In *International Workshop on Data Integration in the Life Sciences*, volume 4075 of *LNCS*, pages 248–263, 2006.

[92] J. Miller, D. Palaniswami, A. Sheth, K. Kochut, and H. Singh. Webwork: METEOR2's web-based workflow management system. *Journal of Intelligent Information Systems*, 10(2):185–215, 1998.

[93] 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*, 7(1):16–27, 1996.

[94] P. Missier, S. Soiland-Reyes, S. Owen, W. Tan, A. Nenadic, I. Dunlop, A. Williams, T. Oinn, and C. Goble. Taverna reloaded. In *International conference on scientific and statistical database management*, pages 471–481, 2010.

[95] L. Moreau, J. Freire, J. Futrelle, R. Mcgrath, J. Myers, and P. Paulson. The open provenance model: An overview. pages 323–326. 2008.

[96] J. Nitzsche, T. van Lessen, D. Karastoyanova, and F. Leymann. BPEL$^{\text{light}}$. In *BPM*, pages 214–229, 2007.

[97] T. Oinn, M. J. Addis, J. Ferris, D. Marvin, M. Senger, T. Carver, M. Greenwood, K. Glover, M. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workows. *Bioinformatics*, 20(17):3045–3054, 2004.

[98] T. Oinn, M. Greenwood, M. Addis, N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. 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*, 18:1067–1100, 2006.

[99] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig Latin: a not-so-foreign language for data processing. In *ACM SIGMOD International Conference on Management of Data*, pages 1099–1110, 2008.

[100] C. Ouyang, E. Verbeek, A. van der, S. Breutel, M. Dumas, and A. ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162–198, 2007.

[101] J. Ram, C. Mller, M. Beckmann, and J. Hardege. The spawning pheromone cysteine-glutathione disulfide ('Nereithione') arouses a momponent nuptial behaviour and electrophysiological activity in Nereis Succinea males. *Federation of American Societies for Experimental Biology*, 13:945–952, 1999.

[102] J. Rasure and C. Williams. An integrated data flow visual language and software development environment. *Journal of Visual Languages and Computing*, 2:217–246, 1991.

[103] D. Roman, H. Lausen, U. Keller, U. Oren, C. Bussler, M. Kifer, and D. Fensel. Web service modeling ontology (WSMO). Retrieved from `http://www.wsmo.org/2004/d2/v1.0/`.

[104] M. Roth, H. Korth, and A. Silberschatz. Extended algebra and calculus for nested relational databases. *ACM Transactions on Database Systems*, 13(4):389–417, 1988.

[105] N. Russell and A. ter Hofstede. newYAWL: Towards workflow 2.0. *Special issue of LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC) II on Concurrency in Process-Aware Information Systems*, 5460:79–97, 2009.

[106] N. Russell, W. van der Aalst, and A. ter Hofstede. Workflow exception patterns. In *International Conference on Advanced Information Systems Engineering*, pages 288–302, 2006.

[107] 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*, pages 104 – 113, 1996.

[108] D. Shukla and B. Schmidt. *Essential Windows Workflow Foundation*. Addison-Wesley Pearson Education, 2007.

[109] H. Sieburg. Physiological studies in silico. *Studies in the Sciences of Complexity*, 12:321–342, 1990.

[110] Y. Simmhan, B. Plale, and D. Gannon. Karma2: Provenance management for data-driven workflows. *International Journal of Web Services Research*, 5(2):1–22, 2008.

[111] M. Sonntag, D. Karastoyanova, and E. Deelman. BPEL4Pegasus: Combining business and scientific workflows. In *International Conference of Distributed Computing Systems*, pages 728–729, 2010.

[112] V. Springel. Simulations of the formation, evolution, and clustering of galaxies and quasars. *Nature*, 435(7042):629–636, 2005.

[113] J. Sroka, J. Hidders, P. Missier, and C. Goble. A formal semantics for the Taverna 2 workflow model. *Journal of Computer and System Sciences*, 76:490–508, 2010.

[114] C. Stefansen. SMAWL: A small workflow language based on CCS. *Center for Advancement of Informal Science Education*, 2005.

[115] W. Tan, P. Missier, I. Foster, R. Madduri, D. Roure, and C. Goble. A comparison of using Taverna and BPEL in building scientific workflows: the case of caGrid. *Concurrency and Computation: Practice and Experience*, 22(9):1098–1117, 2010.

[116] I. Taylor, E. Deelman, D. Gannon, and M. Shields. *Workflows for e-science*. Springer, 2007.

[117] D. Turi, P. Missier, C. Goble, D. Roure, and T. Oinn. Taverna workflows: Syntax and semantics. In *IEEE International Conference on e-Science and Grid Computing*, pages 441–448, 2007.

[118] W. van der Aalst and A. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.

[119] W. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT press, 2002.

[120] W3C. OWL Web Ontology Language Reference. W3C Recommendation, 10 February 2004. M. Dean and G. Schreiber (Eds.). Available from `http://www.w3.org/TR/2004/REC-owl-ref-20040210/`.

[121] Y. Wang and J. Huai. Comparative analysis of BPEL4WS and a Grid workflow language called GPEL. In *International Conference on Services Computing*, pages 253–254, 2005.

[122] Y. Wang and J. Huai. A new Grid workflow description language. In *International Conference on Services Computing*, pages 257–260, 2005.

[123] I. Wassink, H. Rauwerda, P. van der Vet, T. Breit, and A. Nijholt. E-BioFlow: Different perspectives on scientific workflows. In *BIRD*, pages 243–257, 2008.

[124] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WSBPEL, WS-Reliable Messaging and More*. Prentice-Hall, Upper Saddle River, New Jersey, 2005.

[125] K. Weng. Stream oriented computation in recursive data-flow schemas. Technical report, Masters Thesis,Laboratory for Computer Science, Massachusetts Institute if Technology, 1975.

[126] P. Whiting and R. Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16(4):38–59, 1994.

[127] G. Wirtz. Using a visual software engineering language for specifying and analyzing workflows. *Visual Languages*, pages 97–98, 2000.

[128] P. Wong and J. Gibbons. A process-algebraic approach to workflow specification and refinement. In *Software Composition*, pages 51–65, 2007.

[129] Y. Yang, S. Tang, W. Zhang, and L. Fang. A workflow language for grid services in OGSI-based grids. In *Grid and Cooperative Computing*, pages 65–72, 2004.

[130] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *ACM SIGMOD Record*, 34(3):44–49, 2005.

[131] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *USENIX Symposium on Operating Systems*, pages 1–14, 2008.

[132] Y. Zhao, J. Dobson, I. Foster, L. Moreau, and M. Wilde. A notation and system for expressing and executing cleanly typed workflows on messy scientific data. *SIGMOD Record*, 34(3):37–43, 2005.

ABSTRACT

A SCIENTIFIC WORKFLOW FRAMEWORK
FOR
SCIENTIFIC DATA QUERYING AND PROCESSING

by

XUBO FEI

December 2011

Advisor: Dr. Shiyong Lu

Major: Computer Science

Degree: Doctor of Philosophy

We are at the beginning of the new era of "e-science". Researchers in many areas of science, especially in astrophysics, physics, climatology and biology, are now facing tremendous increases in data volumes, as well as corresponding data analysis tools. These increased data and tools demand a better framework to manage the new generation scientific research cycle from data capture, data curation to data analysis, data query and data visualization. Scientific workflows are proving to be one of the key technologies for scientists to formalize and structure complex scientific processes to enable and accelerate many significant scientific discoveries. Although several scientific workflow management systems (SWFMSs) are developed, a formal scientific workflow composition framework, in which workflows and constructs can be composed arbitrarily to process and query collectional scientific data sets, is still to be proposed.

In this thesis, I make several contributions towards formalizing a scientific workflow composition framework. First, We proposed a dataflow-based scientific workflow composition model including a scientific workflow model that separates the declaration of the workflow interface from the definition of its functional body; and a set of workflow constructs, including Map, Reduce, Tree, Loop, Conditional, and Curry, which are fully compositional one

with another. Our workflow composition framework is unique in that workflows are the only operands for composition; in this way, our approach elegantly solves the two-world problem in existing composition frameworks, in which composition needs to deal with both the world of tasks and the world of workflows. Second, We formalized a collection-oriented data model, called collectional data model, to model hierarchical collection-oriented scientific data, and a set of well-defined operators to manipulate and query such data. To our best knowledge, this is the first algebraic approach to modeling collection-oriented scientific data. Finally, we developed a prototype scientific workflow management system, called VIEW. The VIEW system implemented the above techniques in its subsystems and integrated them within a service-oriented architecture.

AUTOBIOGRAPHICAL STATEMENT

Xubo Fei

Xubo Fei is currently working toward the PhD degree in the Department of Computer Science, Wayne State University. He is currently a member of the Scientific Workflow Research Laboratory (the SWR Lab). His current research interests include scientific workflows, scientific data management and their applications in bioinformatics and biology simulation. His research has resulted in several refereed several papers in refereed international journals and conferences.