17

Virtual Data Language: A Typed Workflow Notation for Diversely Structured Scientific Data

Yong Zhao, Michael Wilde, and Ian Foster

17.1 Introduction

When constructing workflows that operate on large and complex data sets, the ability to describe the types of both data sets and workflow procedures can be invaluable, enabling discovery of data sets and procedures, type checking and composition of procedure calls, and iteration over composite data sets.

Such typing should in principle be straightforward because of the hierarchical structure of most scientific data sets. For example, in the functional magnetic resonance imaging (fMRI) applications in cognitive neuroscience research that we use for illustrative purposes in this chapter, we find a hierarchical structure of studies, groups, subjects, experimental runs, and images. A typical application might build a new study by applying a program to each image in each run for each subject in each group in a study.

Unfortunately, we find that such clean logical structures are typically represented in terms of messy physical constructs (e.g., metadata encoded in directory and file names) employed in ad hoc ways. For example, the fMRI physical representation with which we work here is a deeply nested directory structure, with ultimately a single 3D image ("volume") represented by two files located in the same directory, distinguished only by filename suffix. The members of a data set are typically distinguished by identifiers embedded in filenames using diverse, ad hoc conventions.

Such nonuniform physical representations make program development, composition, and execution unnecessarily difficult. While we can incorporate knowledge of file system layouts and file formats into application programs and scripts, the resulting code is hard to write and read, cannot easily be adapted to different representations, and is not clearly typed.

We have previously proposed that these concerns be addressed by separating abstract structure and physical representation [149]. (Woolf et al. [477] have recently proposed similar ideas.) We describe here the design, implementation, and evaluation of a notation that achieves this separation.

We call this notation a *virtual data language* (VDL) because its declarative structure allows data sets to be defined prior to their generation and without regard to their location and representation. For example, consider a VDL procedure "foo_run" with the signature "Run Y=foo_run(Run X)" and with an implementation that builds and returns a new run Y by applying a program "foo" to each image in the run supplied as argument X (X and Y being data set variables of type Run). We can then specify via the VDL procedure invocation "run2=foo_run(run1)" that data set "run2" is to be derived from data set "run1." Independence from location and representation is achieved via the use of XML Data Set Typing and Mapping (XDTM) [303] mechanisms, which allow the types of data sets and procedures to be defined abstractly in terms of XML schema. Separate *mapping descriptors* then define how such abstract data structures translate to physical representations. Such descriptors specify, for example, how to access the physical files associated with "run1" and "run2."

VDL's declarative and typed structure makes it easy to build up increasingly powerful procedures by composition. For example, a procedure "Subject Y = foo_subject(Subject X)" might apply the procedure "foo_run" introduced earlier to each run in a supplied subject. The repeated application of such compositional forms can ultimately define large directed acyclic graphs (DAGs) comprising thousands or even millions of calls to "atomic transformations," each of which operates on just one or two image files.

The expansion of data set definitions expressed in VDL into DAGs, and the execution of these DAGs as workflows in uni- or multiprocessor environments, is the task of an underlying *virtual data system* (VDS) [148], which is comprised of workflow translators, planners, and interfaces to enactment engines.

We have applied our techniques to fMRI data analysis problems [439]. We have modeled a variety of data set types (and their corresponding physical representations) and constructed and executed numerous computational procedures and workflows that operate on those data sets. Quantitative studies of code size in a previous paper [496] suggest that VDL and VDS facilitate easier workflow expression and hence may improve productivity.

This chapter describes:

1. the design of a practical workflow notation and system that separate logical and physical representation to allow the construction of complex workflows on messy data using cleanly typed computational procedures;
2. the VDL type system, as well as the interfaces for mapping specification and program invocation; and
3. a demonstration and evaluation of a prototype of the technology via the encoding and execution of large fMRI workflows in a distributed environment.

This chapter is organized as follows. In Section 17.2, we review related work. In Section 17.3, we introduce the XDTM model, and in Sections 17.4 and

17.5 we describe VDL, using a simplified science application for illustration. In Section 17.6, we apply this model to a real example drawn from procedures used to prepare fMRI study data for analysis. In Section 17.7, we describe our prototype implementation, and in Section 17.8 we conclude with an assessment of this approach.

17.2 Related Work

The Data Format Description Language (DFDL) [42], like XDTM, uses XML schema to describe abstract data models that specify data structures independent from their physical representations. DFDL is concerned with describing legacy data files and complex binary formats, while XDTM focuses on describing data that span files and directories. Thus, the two systems can potentially be used together.

In MIX (Mediation of Information using XML) [40], each data source is also treated as an XML source, and its structural information is represented by an XML DTD. Queries are expressed in a high-level declarative XML query language called XMAS (XML Matching and Structuring Language), which allows object fusion and pattern matching and supports construction of new integrated XML objects from existing ones. MIX's query evaluation takes a virtual approach, where XML queries expressed in XMAS are unfolded and rewritten at runtime and sent to corresponding sources.

The IBM virtual XML garden project [208] provides virtual XML views on diverse data sources such as file systems, zip archives, and databases. It supports XML access and processing on these data sources by writing thin, on-demand adapters that wrap arbitrary data structures into a generic abstract XML interface corresponding to the XML Infoset as well as the XPath and XQuery Data Model.

XML Process Description Language (XPDL) [485], BPEL, and WSDL also use XML schema to describe data or message types but assume that data are represented in XML; in contrast, XDTM can describe "messy" real-world data by mapping from a logical XML view to arbitrary physical representations. Ptolemy [130] and Kepler [19] provide a static typing system; Taverna [326] and Triana [91] do not mandate typing. XDTM's ability to map logical types from/to physical representations is not provided by these languages and systems.

When composing programs into workflows, we must often convert logical types and/or physical representations to make data accessible to downstream programs. XPDL employs scripting languages such as JavaScript to select subcomponents of a data type, and BPEL uses XPath expressions in *Assign* statements for data conversion. VDL permits the declarative specification of a rich set of mapping operations on composite data structures and substructures.

Many workflow languages allow sequential, parallel, and recursive patterns but do not directly support iteration. Taverna relies on its workflow engine to run a process multiple times when a collection is passed to a singleton-argument process. Kepler uses a "map" operator to apply a function that operates on singletons to collections. VDL's typing supports flexible iteration over data sets — and also type checking, composition, and selection.

17.3 XDTM Overview

XDTM defines a data set's *logical structure* via a subset of XML schema, which defines primitive scalar data types, such as Boolean, integer, string, float, and date, and also allows for the definition of complex types via the composition of simple and complex types.

A data set's *physical representation* is defined by a *mapping descriptor*, which defines how each element in the data set's logical schema is stored in, and fetched from, physical structures such as directories, files, and database tables. The original XDTM description [303] indicated that a mapping descriptor groups together a set of mapping functions, each associated with an XML schema type, but did not specify exactly how these mapping functions would be defined. In this chapter, we describe an approach to defining and applying these mapping functions.

In order to permit reuse for different data sets, mapping functions may be parameterized for such things as data set locations. Thus, in order to access a data set, we need to know three things: its type schema, its mapping descriptor, and the value(s) of any parameter(s) associated with the mapping descriptor. These three components are grouped to form a *data set handle*.

Note that multiple mappings may be defined for the same logical schema (i.e., for a single logical type). For example, an array of numbers might be physically represented, in different contexts, as a set of relations, a text file, a spreadsheet, or an XML document.

17.4 Physical and Logical Structure: An Example

We use a simple example to illustrate the relationship between physical and logical structure. This example concerns the analysis of data collected from portable cosmic ray detectors. Such detectors are increasingly used in secondary-level physics education through projects such as QuarkNet [36].

As in many scientific experiments, the nature of the data collection process determines the data's physical representation. Students are organized into groups; each group installs a few detectors and collects data from the detectors. Data from detectors are sent to PCs in the form of simple text files that describe the sampling of A/D converter levels on the multiple channels of the instrument (Figure 17.1). (We can think of these "raw data files"

as describing potential cosmic ray events in the form of digitized waveform descriptions.) Analysis then consists of processing these raw waveforms to eliminate noise, extracting a signal, and then searching for correlations in the data from multiple channels, multiple instruments at varying locations, and multiple runs.

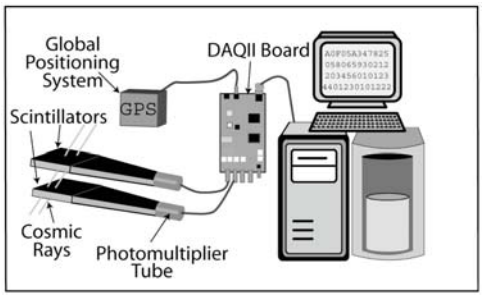

Figure 17.1: Cosmic ray detector.

As depicted in Figure 17.2, a suitable physical data set organization for this application is a hierarchical directory structure that provides for multiple experimental groups, each with data from one or more *detectors*. (Note that directories are distinguished from files by a trailing "/".) Observations consist of *raw data* from the instruments along with metadata about the time period of the recording and the physical location and orientation of the detector ("*geometry*"). One metadata file per detector ("*detector_info*") is also present in the structure. Derived data produced via various data analysis procedures are stored in the same structure. *Pulse* files are an example of an output data set added to the observation following the application of a reconstruction procedure to the raw events.

To illustrate how "messy" a physical representation can be, consider that in this application we could represent the start date/time of an observation using the creation time of the *rawdata* file and the end time of the observation by the modification time of that file.

In contrast to these ad hoc physical encodings, the *logical* structure of such a physical data set representation can be uniformly and explicitly described by XML schema, as illustrated in Figure 17.3.

17.5 Virtual Data Language

XDTM specifies how we define XML structures and associate physical representations with them. However, it does not address how we write programs that operate on XDTM-defined data. That is the focus of the

XDTM-based Virtual Data Language (VDL). This language, derived loosely from an earlier VDL [148] that dealt solely with untyped files, allows users to define procedures that accept, return, and operate on data sets with type, representation, and location defined by XDTM. We introduce the principal features of VDL via a tutorial example.

17.5.1 Representing Logical Structure: The VDL Type System

VDL uses a C-like syntax to represent XML schema complex types, as illustrated in Figure 17.4, which shows VDL type definitions corresponding to the XML schema of Figure 17.3. The *Detector* type contains information about the detector hardware — such as serial number, installation date, and firmware revision (*DetectorInfo*) — and a set of *Observations*. Each *Observation* contains the time range for which the raw data are gathered (*ostart*, *oend*), the raw data themselves, some geometry information, and a derived data type *Pulse*. The conversion from this notation to XML schema is straightforward: The VDL data model of named member fields ("structures" or "records") and arrays is mapped to XML schema constructs for sequences and element cardinality (occurrence).

17.5.2 Accessing Physical Structure: Mapping Functions

The process of *mapping*, as defined by XDTM, converts between a data set's physical representation (typically in persistent storage) and a logical XML view of those data. VDL programs operate on this logical view, and *mapping functions* implement the actions used to convert back and forth between the XML view and the physical representation.

Associated with each logical type is a mapping descriptor, which provides access to a set of mapping functions that the VDL implementation may invoke during program execution. A mapping descriptor must include the following four functions:

```
/quarknet/
/quarknet/group1/
/quarknet/group1/detector1/
/quarknet/group1/detector1/detector_info
/quarknet/group1/detector1/observation1/
/quarknet/group1/detector1/observation1/geometry
/quarknet/group1/detector1/observation1/rawdata
/quarknet/group1/detector1/observation1/pulse
/quarkent/group1/detector1/observation2/
...
/quarknet/group1/detector2/
...
```

Figure 17.2: Physical directory structure of detector data.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema targetNamespace="http://quarknet.org/schema/cosmic.xsd"
           xmlns="http://quarknet.org/schema/cosmic.xsd"
           xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:complexType name="Observation">
    <xs:sequence>
      <xs:element name="ostart" type="xs:date"/>
      <xs:element name="oend" type="xs:date"/>
      <xs:element name="rawdata" type="RawData"/>
      <xs:element name="geo" type="Geometry"/>
      <xs:element name="pulse" type="Pulse"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="Detector">
    <xs:sequence>
      <xs:element name="info" type="DetectorInfo"/>
      <xs:sequence minOccurs="0" maxOccurs="unbounded">
        <xs:element name="ob" type="Observation"/>
      </xs:sequence>
    </xs:sequence>
  </xs:complexType>

</xs:schema>
```

Figure 17.3: XML schema of detector data.

```
type Detector {
    DetectorInfo info;
    Observation ob[ ];
}

type Observation {
    Date ostart, oend;
    RawData rawdata;
    Geometry geo;
    Pulse pulse; /* a derived file: pulses reconstructed from raw */
}

type DetectorInfo {
    Int serialNum;
    Date installDate;
    String swRev;
}
```

Figure 17.4: VDL type definition for detector data.

- create data set: creates a physical data set conforming to the desired physical representation;
- store member: stores a specific element of the logical structure into its physical storage;
- get member: gets a specific logical child element from the data set's physical storage;
- get member list: gets a list of child elements from the physical storage.

In addition, a mapping descriptor often includes additional mapping functions that provide access to various components of the physical data representation. For example:

- filename: provides access to the file or directory name of a data set member's physical representation.

To realize the mapping model in VDL, we formalize the concept of the XDTM logical XML view by defining a construct much like an XML store [435], which we call the *xview*. The xview is managed by the VDL runtime implementation, which we refer to abstractly as the *virtual data machine*, or VDM.

As a VDL program executes, the VDM performs VDL expression evaluation by invoking the appropriate mapping functions to move data back and forth between physical data representations and the xview. When a mapping function maps a data set's representation into the xview, it creates the XML representation of the physical data structure, in which each field (or member) of a data set type becomes either an atomic value or a handle to the corresponding physical data set component. VDL variables that are defined as local to a VDL procedure (i.e., "stack variables" in a procedure's activation record [363]) are represented in a similar manner.

The xview can be implemented in many ways (for instance, in an XML database) and has the desirable features that (a) it can be processed by standard XML tools such as XPath and XQuery and (b) it can operate as a cache, logically representing an entire physical data set but physically "faulting in" data sets as they are referenced in a "lazy evaluation" mode and swapping out data sets that are not currently being referenced on a least recently used basis.

17.5.3 Procedures

Data Sets are operated on by *procedures*, which take data sets described by XDTM as input, perform computations on those data sets, and produce data sets described by XDTM as output.

A VDL procedure can be either an *atomic procedure* or a named workflow template defining a DAG of multiple nodes (*compound procedure*). An atomic procedure definition specifies an interface to an executable program or service (more on this topic below), while a compound procedure composes calls to atomic procedures, other compound procedures, and/or control statements.

VDL *procedure definitions* specify formal parameters; procedure calls supply the actual argument values. For example, consider an atomic procedure *reconstruct* that performs reconstruction of pulse data from raw data. The following interface specifies that this procedure takes Raw and Geometry as input data set types and produces Pulse as an output type:

```
(Pulse pulse ) reconstruct ( Raw raw, Geometry geo) {
    /* procedure body */
}
```

We present the corresponding procedure body in Section 17.5.4.

17.5.4 Atomic Procedures

An atomic procedure defines an interface to an external executable program or Web service and specifies how logical data types are mapped to and from application program or service arguments and results.

Invoking Application Programs

An atomic procedure that defines an interface to an external program must specify:

- The interpreter that will be used to execute the program
- The name of the program to be executed
- The syntax of the command line that must be created to invoke the program
- The data context in which the program is to execute (i.e., the physical data sets that need to be available to the program when it executes)

An atomic procedure has a header that specifies its interface in terms of data set types, but its body operates on data set representations. Thus, expressions in the data set body must be able to use mapping functions (Section 17.5.2) to map between types and representations.

The header of an atomic procedure that defines an interface to an external program specifies the name of the program to be invoked (the atomic procedure call), the data to be passed to the procedure's execution site (the atomic procedure's input arguments), and the resulting data to be returned back from the procedure's execution site (the atomic procedure's return arguments).

The body of such an atomic procedure specifies how to set up its execution environment and how to assemble the call to the procedure. For example, the following procedure *reconstruct* defines a VDL interface to a cosmic ray data-processing application of the same name. The statements in the body assemble the command to invoke the program, with the "bash" statement indicating that invocation is to occur by using the bash shell command interpreter. The @ notation is used to invoke a mapping function. In this example, the mapping function "filename" is called to extract filenames from the data set representation so they can be passed to the shell.

```
(Pulse pulse ) reconstruct ( Raw raw, Geometry geo) {
    bash {
      reconstruct
        @filename( raw )
        @filename( geo )
        @filename( pulse )
    }
}
```

This atomic procedure may be invoked by a procedure call such as

```
Pulse p1 = reconstruct ( raw1, geo );
```

which takes a raw data set *raw1* and a geometry data set *geo* as inputs and generates as output a pulse data set *p1*. The data sets *raw1*, *geo*, and *p1* are defined as data set handles, which include the typing and mapping specifications for these data sets.

```
RawData  raw1 <file_mapper;
                location="/quarknet/group1/detector1/observation1/rawdata">
Geometry geo  <file_mapper;
                location="/quarknet/group1/detector1/observation1/geometry">
Pulse    p1   <file_mapper;
                location="/quarknet/group1/detector1/processed/pulse/p1">
```

The procedure call is compiled into the execution of the following command line:

```
reconstruct    /quarknet/group1/detector1/observation1/rawdata \
               /quarknet/group1/detector1/observation1/geometry \
               /quarknet/group1/detector1/processed/pulse/p1
```

If this command is executed on a remote host that does not share a file system, then VDS must ensure that the physical representations of data sets passed as input arguments are transferred to the remote site, enabling the executing application to access the required physical files. For example, in the call just shown, the physical representations of the data sets *raw1* and *geo* must be transferred to the remote site.

Similarly, output data (e.g., *p1* in the example call) must be made accessible to other program components. To this end, the existence of the physical data on the remote site is recorded. In addition, the data are optionally copied to a specified site to create an additional replica (which often serves as an archival copy). Finally, the xview itself must be updated to be brought back in sync with the physical representation.

Invoking Web Services

We envision that atomic procedure definitions could also specify Web service interfaces. Such procedures would have the same procedure prototype header as an application program interface but provide a different body. The following example defines a Web service implementation of the same *reconstruct* procedure that was defined above as an executable application.

```
(Pulse pulse) reconstruct (Raw raw, Geometry geo)
{
    service {
        wsdlURI = "http://quarknet.org/cosmic.wsdl";
        portType = "detectorPT";
        operation = "reconstruct";
        soapRequestMsg  = { rawdata = raw;
                geometry = geo};
        soapResponseMsg = { pulsedata = pulse};
    }
}
```

Not shown here is the specification of how arguments such as raw, geo, and pulse are to be passed to and from the Web service. For this, data transport options such as the following will be required:

1. File reference: A reference to a file is passed in the Web service message in the form of a URI.
2. File content: The content of a file is encoded into an XML element and passed in the message body.
3. SOAP attachment: The content of a file is passed as a SOAP attachment.

17.5.5 Compound Procedures

A compound procedure contains a set of calls to other procedures. Variables in the body of a compound procedure specify data dependencies and thus the directed arcs for the DAG corresponding to the compound procedure's workflow. For example:

```
(Shower s) showerstudy (Observation o1, Observation o2) {
    Pulse p1 = thresholdPulse (o1.pulse);
    Pulse p2 = thresholdPulse (o2.pulse);
    Pulse p = correlateEvents (p1, p2);
    s = selectEvents (p);
}
```

In the procedure *showerstudy*, which computes the correlation between two observations, the pulse events from each observation are first filtered by a thresholding procedure, then the results of the thresholding procedures are combined by a correlation procedure, and finally interesting shower events are selected from the combined events. In this compound procedure, data dependencies dictate that the two invocations of *thresholdPulse* can be executed in parallel, after which the calls to *correlateEvents* and *selectEvents* must execute in sequence.

Arbitrary workflow DAGs can be specified in this manner, with the nodes of the DAGs being procedure calls and the edges represented by variables, which are employed to pass the output of one procedure to the input of another.

17.5.6 Control-Flow Constructs

Control-flow constructs are special control entities in a workflow that control the direction of execution. VDL provides *if*, *switch*, *foreach*, and *while* constructs, with syntax and semantics similar to comparable constructs in high-level languages. We illustrate the use of the *foreach* construct in the following example:

```
genPulses ( Detector det ) {
    foreach Observation o in det.ob {
        o.pulse = reconstruct (o.raw, o.geo);
    }
}
```

This example applies the atomic procedure *reconstruct* to each of the observations associated with a specific detector *det* and generates the pulse data for each observation from the raw data. All of the calls to *reconstruct* can be scheduled to run in parallel.

17.6 An Application Example: Functional MRI

VDL provides an effective way to specify the preprocessing and analysis of the terabytes of data contained in scientific archives such as the fMRI Data Center (fMRIDC: `www.fmridc.org`), based at Dartmouth College. This publicly available repository includes complete data sets from published studies of human cognition using functional magnetic resonance imaging (fMRI). Data Sets include 4D functional image-volume time-course data, high-resolution images of brain anatomy, study metadata, and other supporting data collected as part of the study. The fMRIDC curates and packages these data sets for open dissemination to researchers around the world, who may use the data to conduct novel analyses, test alternative hypotheses, explore new means of data visualization, or for education and training.

17.6.1 Overview of fMRI Data Sets

fMRI data sets are derived by scanning the brains of subjects as they perform cognitive or manual tasks. The raw data for a typical study might consist of three subject groups with 20 subjects per group, five experimental runs per subject, and 300 volume images per run, yielding 90,000 volumes and over 60 GB of data. A fully processed and analyzed study data set can contain over 1.2 million files. In a typical year at the Dartmouth Brain Imaging Center, about 60 researchers preprocess and analyze about 20 concurrent studies.

Experimental subjects are scanned once to obtain a high-resolution image of their brain anatomy ("anatomical volume") and then scanned with lower resolution at rapid intervals to observe the effects of blood flow from the "BOLD" (blood oxygenated level dependent) signal while performing some task ("functional runs"). These images are preprocessed and subjected to intensive analysis that begins with image processing and concludes with a statistical analysis of correlations between stimuli and neural activity.

Figure 17.5 illustrates some of the conventions that are frequently used in the physical representation of such fMRI data sets. The logical representation on the left shows the hierarchy of objects in a hypothetical study, while the physical representation on the right indicates the typical manner in which the objects in the logical view are physically represented in a hierarchical file system directory, making heavy use of the encoding of object identifiers into the names of files and directories.

The VDL examples in the next subsections are based on a workflow, AIRSN, that performs *spatial normalization* for preprocessing raw fMRI data prior to analysis. AIRSN normalizes sets of time series of 3D volumes to a standardized coordinate system and applies motion correction and Gaussian smoothing.

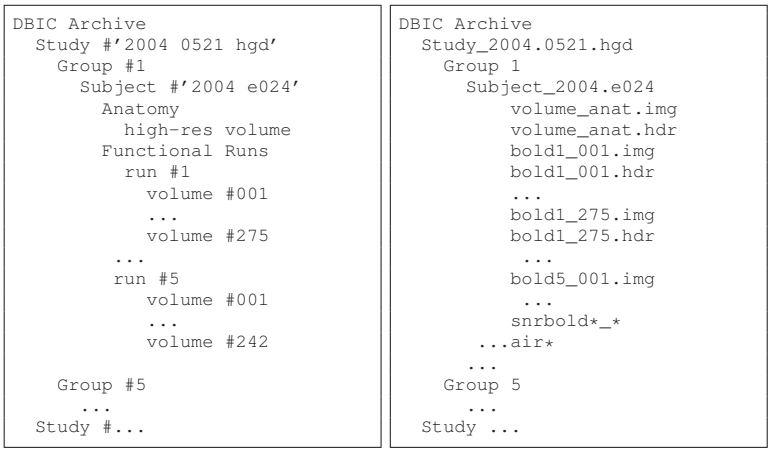

Figure 17.5: fMRI structure — logical (left) and physical (right).

17.6.2 fMRI Data Set Type Definitions

Figure 17.6 shows the VDL types that represent the data objects of Figure 17.5. A *Volume* contains a 3D image of a volumetric slice of a brain image, represented by an *Image* (voxels) and a *Header* (scanner metadata). As we do not manipulate the contents of those objects directly within this VDL program, we do not further decompose their structure. A time series of volumes taken from a functional scan of one subject, doing one task, forms a *Run*. In typical experiments, each *Subject* has an anatomical data set, *Anat*, and multiple input and normalized runs.

Specific output formats involved in processing raw input volumes and runs may include outputs from various image-processing tools, such as the automated image registration (AIR) suite [475]. The type *Air* corresponds to one of the data set types created by these tools (and it, too, needs no finer decomposition).

```
type Volume { Image img; Header hdr; }
type Run { Volume v[ ]; }
type Anat Volume;
type Subject { Anat anat; Run run [ ]; Run snrun [ ]; }
type Group { Subject s[ ]; }
type Study { Group g[ ]; }
type AirVector { Air a[ ]; }
type NormAnat { Anat aVol; Warp aWarp; Volume nHires; }
```

Figure 17.6: VDL type definition for fMRI data.

17.6.3 VDL Procedures for AIRSN

Figure 17.7 shows a subset of the VDL procedures for AIRSN. The procedure *functional()* expresses the steps in Figure 17.8; *airsn_subject()* calls this procedure once per each component and *anatomical()* (not shown) to process a *Subject*. *airsn_subject()* creates in the *Subject* data set a new spatially normalized *Run* for each raw *Run*. Such procedures define how the workflow is expanded to process the members of a data set, as in Figure 17.9.

```
(Run snr) functional( Run r, NormAnat a, Air shrink ) {
    Run yroRun = reorientRun( r , "y" );
    Run roRun = reorientRun( yroRun , "x" );
    Volume std = roRun[0];
    Run rndr = random_select( roRun, .1 ); //10% sample
    AirVector rndAirVec = align_linearRun( rndr, std, 12, 1000, 1000, [81,3,3] );
    Run reslicedRndr = resliceRun( rndr, rndAirVec, "o", "k" );
    Volume meanRand = softmean( reslicedRndr, "y", null );
    Air mnQAAir = alignlinear( a.nHires, meanRand, 6, 1000, 4, [81,3,3] );
    Volume mnQA = reslice( meanRand, mnQAAir, "o", "k" );
    Warp boldNormWarp = combinewarp( shrink, a.aWarp, mnQAAir );
    Run nr = reslice_warp_run( boldNormWarp, roRun );
    Volume meanAll = strictmean ( nr, "y", null )
    Volume boldMask = binarize( meanAll, "y" );
    snr = gsmoothRun( nr, boldMask, 6, 6, 6 );
}

airsn_subject( Subject s, Volume atlas, Air ashrink, Air fshrink ) {
    NormAnat a = anatomical( s.anat, atlas, ashrink );
    Run r, snr;
        int i;
        foreach (r,i) in s.run {
                snr = functional( r, a, fshrink );
                s.snrun[ i ] = snr;
        }
}
```

Figure 17.7: VDL fMRI procedure examples.

17.6.4 The AIRSN Workflow

Figures 17.8 and 17.9 show two views of the most data-intensive segment of the AIRSN workflow, which processes the data from the functional runs of a study. Figure 17.8 is a high-level representation in which each oval represents an operation performed on an entire *Run*. Figure 17.9 expands the workflow to the *Volume* level for a data set of ten functional volumes. (Note that the *random_select* call is omitted in Figure 17.9). In realistic fMRI experiments, *Runs* might include hundreds or thousands of *Volumes*.

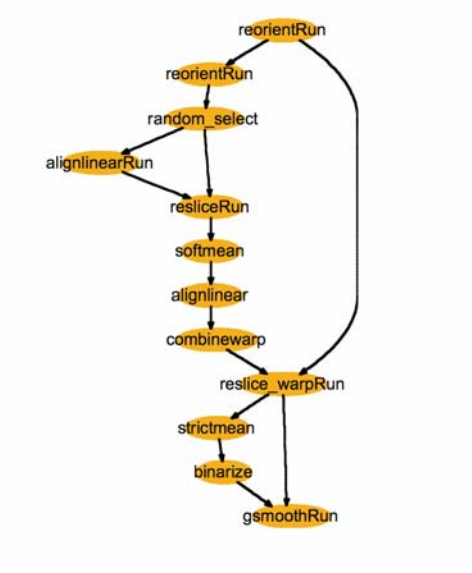

Figure 17.8: AIRSN workflow high-level representation.

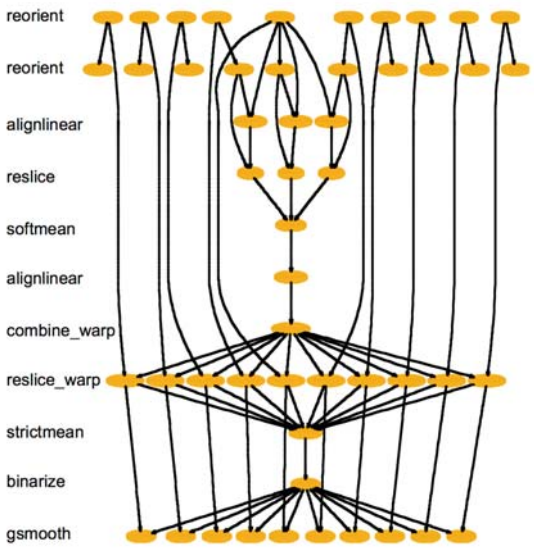

Figure 17.9: AIRSN workflow expanded to show all atomic file operations for a ten volume run.

17.7 VDL Implementation

We have developed a prototype system that can process VDL type definitions and mappings, convert a typed workflow definition into an executable DAG, expand DAG nodes dynamically to process subcomponents of a compound data set, and submit and execute the resulting DAG in a Grid environment. The separation of data set type and physical representation that we achieve with VDL can facilitate various runtime optimizations and graph-rewriting operations [112]. The prototype implements the runtime operations needed to support typed VDL data set processing and execution, which is the principal technical challenge of implementing VDL. We have also verified that we can invoke equivalent services and applications from the same VDL.

The prototype extends an earlier VDS implementation with features to support data typing and mapping. We use the VDS graph traversal mechanism to generate an abstract DAG in which transformations are not yet tied to specific applications or services and data objects are not yet bound to specific locations and physical representations. The extended VDS also enhances the DAG representation by introducing "foreach" nodes (in addition to the existing "atomic" nodes) to represent *foreach* statements in a VDL procedure. These nodes are expanded at runtime (see Section 17.7.2), thus enabling data sets to have a dynamically determined size.

The resulting concrete DAG is executed by the DAGMan ("DAG manager") tool. DAGMan provides many necessary facilities for workflow execution, such as logging, job status monitoring, workflow persistence, and recursive fault recovery. DAGMan submits jobs to Grid sites via the Globus GRAM protocol.

17.7.1 Data Mapping

Our prototype employs a table-driven approach to implement XDTM mapping for data sets stored on file systems. Each table entry specifies

```
name:    the data object name
pattern: the pattern used to match filenames
mode:FILE (find matches in directory)
    RLS (find matches via replica location service)
    ENUM (data set content is enumerated)
content: used in ENUM mode to list content
```

When mapping an input data set, this table is consulted, the pattern is used to match a directory or replica location service according to the mode, and the members of the data set are enumerated in an in-memory structure that models the behavior of the xview. This structure is then used to expand *foreach* statements and to set command-line arguments.

For example, in Figure 17.5, a *Volume* is physically represented as an image/header file pair, and a *Run* as a set of such pairs. Furthermore, multiple *Runs* may be stored in the same directory, with different *Runs* distinguished by a prefix and different *Volumes* by a suffix. To map this representation to

the logical *Run* structure, the pattern "bold$N$∗" is used to identify all pairs in Run N at a specified location. Thus, the mapper, when applied to the following eight files, identifies two runs, one with three *Volumes* (*Run 1*) and the other with one (*Run 2*).

```
bold1_001.img    bold1_001.hdr
bold1_002.img    bold1_002.hdr
bold1_003.img    bold1_003.hdr
bold2_007.img    bold2_007.hdr
```

17.7.2 Dynamic Node Expansion

A node containing a *foreach* statement must be expanded dynamically into a set of nodes: one for each member of the target data set specified in the *foreach*. This expansion is performed at runtime: When a *foreach* node is scheduled for execution, the appropriate mapper function is called on the specified data set to determine its members, and for each member of the data set identified (e.g., for each *Volume* in a *Run*), a new job is created in a "sub-DAG."

The new sub-DAG is submitted for execution, and the main job waits for the sub-DAG to finish before proceeding. A postscript for the main job takes care of the transfer and registration of all output files and the collection of those files into the output data set. This workflow expansion process may recurse further if the subcomponents themselves also include *foreach* statements. DAGMan provides workflow persistence in the event of system failures during recursion.

The process of dynamic node expansion can be performed in a cursor-like manner to efficiently navigate large data sets. Large data sets behave *as if* the entire data set is expanded in the xview. A naïve implementation would do exactly that, but a more sophisticated implementation can yield better performance by taking advantage of operations that "close" members after they are mapped and that scroll through large sequences of members in a cursor-like fashion to enable arbitrarily large data sets to be mapped.

17.7.3 Optimizations and Graph Transformation

Since data set mapping and node expansion are carried out at runtime, we can use graph transformations to apply optimization strategies. For example, in the AIRSN workflow, some processes, such as the *reorient* of a single *Volume*, only take a few seconds to run. It is inefficient to schedule a distinct process for each *Volume* in a *Run*. Rather, we can combine multiple such processes to run as a single job, thus reducing scheduling and queuing overhead.

As a second example, the *softmean* procedure computes the mean of all *Volumes* in a *Run*. For a data set with a large number of *Volumes*, this stage is a bottleneck, as no parallelism is engaged. There is also a practical issue: The executable takes all *Volume* filenames as command-line arguments, which can exceed limits defined by the Condor and UNIX shell tools used within our

VDS implementation. Thus, we transform this node into a tree in which leaf nodes compute over subsets of the data set. The process repeats until we get a single output. The shape of this tree can be tuned according to the available computing nodes and data set sizes to achieve optimal parallelism and avoid command-line length limitations.

17.8 Conclusion

We have designed a typed workflow notation and system that allows workflows to be expressed in terms of declarative procedures that operate on XML data types and are then executed on diverse physical representations and distributed computers. We have shown, via studies that compare program sizes with and without our notation [496], that this notation and system can be used to express large amounts of distributed computation easily.

The productivity leverage of this approach is apparent: A small group of developers can define VDL interfaces to the application programs and utilities used in a scientific domain and then create a library of data set types, mappers, and data set iteration functions. Such a "virtual data library" encapsulates low-level details concerning how data are grouped, transported, cataloged, passed to applications, and collected as results. Other scientists can then use such libraries to construct workflows without needing to understand the details of physical representation and furthermore are protected by the XDTM type system from forming workflows that are not type compliant. The data management conventions of a research group can be encoded and uniformly maintained with XDTM mapping functions, thus making it easier to curate data set collections that may include many tens of millions of files.

We next plan to automate the compilation steps that were performed manually in our prototype and to create a complete workflow development and execution environment for our XDTM-based VDL. We will also investigate support for services, automation of type coercions between differing physical representations, and recording of provenance for large data collections.

Acknowledgments

This work was supported by the National Science Foundation GriPhyN Project, grant ITR-800864; the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy; and the National Institutes of Health, grants NS37470 and NS44393. We are grateful to Jed Dobson and Scott Grafton of the Dartmouth Brain Imaging Center, and to our colleagues on the Virtual Data System team, Ewa Deelman, Carl Kesselman, Gaurang Mehta, Doug Scheftner, Karan Vahi, and Jens Voeckler, for discussion, guidance, and assistance.