Collection-Oriented Scientific Workflows for Integrating and Analyzing Biological Data*

Timothy McPhillips[1], Shawn Bowers[1], Bertram Ludäscher[1,2]

[1]UC Davis Genome Center, University of California, Davis
[2]Department of Computer Science, University of California, Davis
{tmcphillips,sbowers,ludaesch}@ucdavis.edu

Abstract. Steps in scientific workflows often generate collections of results, causing the data flowing through workflows to become increasingly nested. Because conventional workflow components (or actors) typically operate on simple or application-specific data types, additional actors often are required to manage these nested data collections. As a result, conventional workflows become increasingly complex as data becomes more nested. This paper describes a new paradigm for developing scientific workflows that transparently manages nested data collections. Collection-oriented workflows have a number of advantages over conventional approaches including simpler workflow designs (*e.g.*, requiring fewer actors and control-flow constructs) that are invariant under changes in data nesting. Our implementation within the Kepler scientific workflow system enables the explicit representation of collections and collection schemas, concurrent operation over collection contents via multi-level pipeline parallelism, and allows collection-aware actors to be composed readily from conventional actors.

1 Introduction

Scientists today require access to data from diverse sources. Nowhere is this need more pressing than in the life sciences, where multiplying databases and rapidly growing data repositories promise to provide researchers with a wealth of information relevant to the systems they study. Effectively exploiting diverse sources of data requires a spectrum of data integration approaches.

In the database community, data integration traditionally means resolving different data structures that represent fundamentally the *same* kind of information [11]. This information may be stored using heterogeneous schemas, and may use different representations for data values (*e.g.*, for identifying objects). In such cases, data integration involves determining mappings between source schemas, and then transforming these schemas into a common schema and corresponding integrated data set that can be used for some other purpose. These mappings and transformations typically represent logically necessary relationships between different data sources.

In contrast, data integration in the life sciences often entails applying fundamentally *different* kinds of information to answer scientific questions, make discoveries,

* Work supported in part by SciDAC/SDM (DE-FC02-01ER25486), NSF/SEEK (DBI-0533368), and NSF/GEON (EAR-0225673)

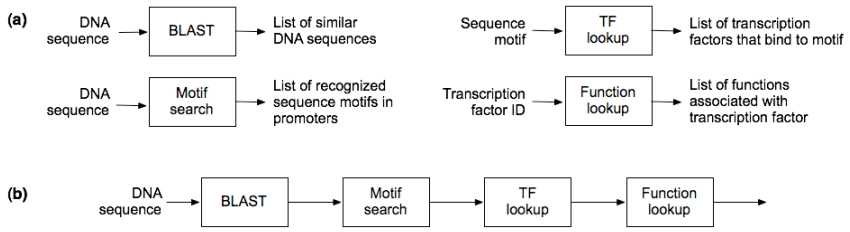

Fig. 1. Scientific workflow components frequently produce lists of results: (a) typical bioinformatics components; and (b) a hypothetical workflow composed from these components that leads to increasingly nested data collections

and test theories. Such scientific data integration procedures necessarily invoke scientific theories that cannot be inferred from schemas or data alone. For example, consider a systematist who wishes to use both genomic sequence data and morphological data in the process of inferring the evolutionary relationships among organisms. Instead of simply mapping DNA sequences and morphological data into a uniform data format, different processes may be applied to each data source to infer evolutionary (*i.e.*, phylogenetic) trees. The systematist then may use the assumption that the organisms have only one true set of evolutionary relationships, and that the phylogenetic trees inferred from genomic and morphological data approximate the true relationships. By employing this theory, the researcher may "integrate" these distinct data sources by computing a consensus tree that reflects commonalities in the distinct phylogenetic trees inferred from the different data sources. These consensus trees (*i.e.*, the resulting data product of integration) can then be analyzed further or applied in other studies.

The challenge of integrating life-science data from multiple sources becomes even more daunting as disciplines become increasingly specialized and as more diverse types of scientific data are desired. Scientific workflow systems [12,13,15,20,22,4] aim at facilitating these types of integration and analysis.[1] However, current scientific workflow systems still offer little or no support for effectively managing (and hiding) the inherent complexity of life-science data, leading to overly complex workflows that are hard to create, reuse, and optimize.

As shown in Figure 1, scientific workflow components (or actors) frequently generate lists of results. When carried out one after the other, such operations naturally yield increasingly nested collections of data that must be managed during workflow execution. This situation is further complicated by the fact that the steps in such workflows in general operate on different nesting levels. For example, a query of a database mapping sequence motifs to known transcription factors might take a single motif as an input, while the operation upstream of this step in the workflow might generate a list of motifs to operate upon. Similarly, the collection of all transcription factors associated with a number of different sequence motifs might be required as input to a downstream component. As these examples demonstrate, scientific workflows must be able to main-

[1] Figure 4 shows an implementation of a workflow for inferring and analyzing phylogenetic trees using the KEPLER system.

tain associations within and between nested lists of intermediate results throughout the workflow, while at the same time presenting to each workflow component data inputs of the correct type and granularity.

We address this problem by proposing a framework for representing and managing nested collections in scientific workflows (Section 2). Our approach is inspired by flow-based programming [18] and techniques used in collection-based [3] and functional programming languages. We represent nested data collections as "flat" sequences of data tokens embedded with special control tokens for delimiting the beginning and end of each collection. We previously have described [17] how our implementation of this approach within the Kepler scientific workflow system provides convenient high-level operations for managing nested collections; facilitates highly pipelined execution of actors operating at different levels of collection nesting; simplifies workflow design; enables context-dependent, dynamic configuration of actors; and supports robust workflow exception handling.

In this paper we define an abstract data model for collection-oriented workflows (Section 3). Using this abstract data model, we then define a lightweight schema language for restricting collection-oriented structures. Collection schemas can be used for a number of purposes. They allow developers to "publish" reusable collection definitions. Schemas are also used in defining *scope* parameters (Section 4), which declare the type of data an actor operates over, and how the actor should be invoked over that data. In general, scope parameters are declarative expressions used to configure collection-aware actors and to simplify actor development. Finally, we introduce an approach that allows collection-aware actors to be composed readily from conventional Kepler actors, and show how this approach can simplify the development of new collection-aware actors and further facilitate reusability in scientific workflows.

2 The Collection-Oriented Workflow Approach

2.1 The Kepler Scientific Workflow System

The Kepler scientific workflow system [1,12] is being developed jointly by a collaboration of application-oriented scientific research projects.[2] Kepler extends the Ptolemy II[3] system (hereafter, Ptolemy) with new features and components for scientific workflow design and for efficient workflow execution using distributed computational and experimental resources. Ptolemy was originally developed as a modeling and simulation environment, *e.g.* to study complex computation models and embedded system applications.

In Kepler, users develop workflows by selecting appropriate components (called *actors*) and placing them on the design canvas. Once on the canvas, components can be "wired" together to form the desired dataflow graph, *e.g.*, as shown in Figure 4. Actors have *input ports* and *output ports* that provide the communication interface to other actors. Workflows can be hierarchically defined, using *composite actors* to contain subworkflows. Control-flow elements such as branches and loops are also supported.

[2] http://www.kepler-project.org/
[3] http://ptolemy.eecs.berkeley.edu/

In KEPLER, actors can be written directly in Java or can wrap external components. For example, KEPLER provides mechanisms to create actors from web services, C/C++ applications, scripting languages, R[4] and Matlab, database queries, SRB[5] commands, and so on.

In KEPLER, data is represented as a sequence of *tokens*, which are passed from one actor to another via actor connections. KEPLER differs from other scientific workflow systems in that the overall execution and component interaction semantics of a workflow is not determined by actors, but instead is defined by a separate component called a *director*. This separation allows actors to be reused in workflows having different models of computation. KEPLER (via PTOLEMY) includes directors that specify, *e.g.*, process network (PN), synchronous dataflow (SDF), continuous time (CT), discrete event (DE), and finite state machine (FSM) computation models.

Most scientific workflows defined using KEPLER use the PN director (based on [9]), or SDF, a restricted version of PN. The PN director executes each actor in a workflow as a separate process (or thread). Actors communicate asynchronously in process networks through buffered channels implemented as queues of effectively unbounded size. The PN director can be used to pipeline data tokens through scientific workflows, enabling highly concurrent execution. In SDF, actors *a priori* define fixed token consumption and production rates. This allows the SDF director to statically schedule actors [10], while guaranteeing certain properties of workflows. PTOLEMY's support for composite actors allow multiple computing models to be used within a single workflow by optionally specifying distinct directors for particular subworkflows, *e.g.*, the PaupHSearch composite actor employs the SDF director (Figure 8), but may be used within a workflow based on the PN director (Figure 4).

2.2 Managing Nested Data Collections in Kepler

KEPLER currently does not provide explicit support for managing nested collections, and workflow authors use a variety of approaches to add this support to KEPLER workflows. The general approach used to support nested collections in KEPLER is shown in Figure 2. Figure 2 (a) shows two conventional KEPLER actors A and B, where the output of A is connected to the input of B. Here, A produces singleton data items of type β (where individual items are denoted β_1, β_2, etc.), which are directly consumed by B. Figure 2 (b) shows a similar workflow, but where actor A has been replaced by actor A′, which produces lists of items of type β instead of only singleton β values. The block labeled *CF* indicates where special control-flow actors are used to unpack and repack list elements.[6] Figure 2 (c) uses the same underlying workflow; in this case however, actor A′ receives a list of input values, introducing additional control-flow blocks. Figure 2 (d) shows the case where an actor A″ produces pairs of items of type (δ, β), the β items are routed using a control-flow bock to the B actor (which expects only β items), and the δ items are routed downstream where they are paired with B's output (again, using a control-flow block) and passed as input to the C actor.

[4] http://www.r-project.org/

[5] Storage Resource Broker, http://www.sdsc.edu/srb/

[6] Control-flow blocks are implemented in a number of ways in practice, but are typically modeled using multiple low-level actors possibly placed within a composite.

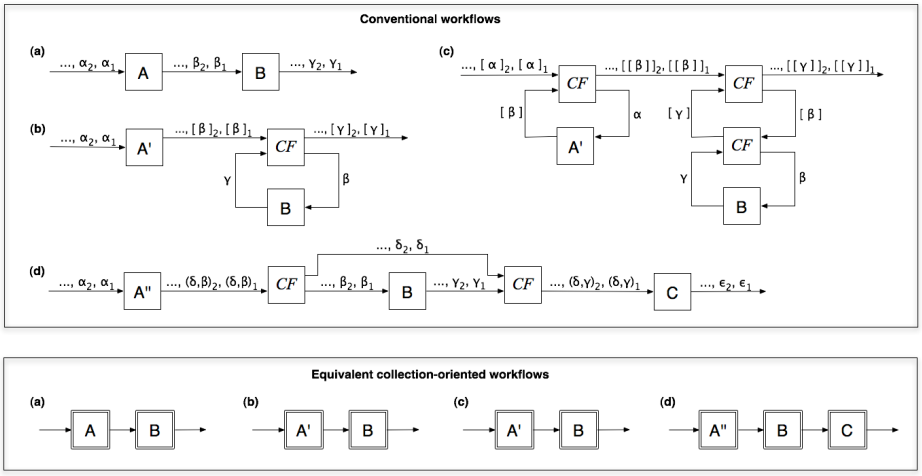

Fig. 2. Conventional scientific workflows with control-flow constructs for handling complex data (top), and corresponding collection-oriented workflows (bottom) in which the control-flow is managed explicitly by the framework

As Figure 2 demonstrates, a significant weakness of using special actors to manage collections is that the resulting workflows must be modified to handle changes in (upstream) data nesting. In principle, one could tailor variants of actors A–C to support particular collection structures, *e.g.*, by embedding the logic represented by the *CF* blocks within custom code in each actor. This approach, however, limits the ability to reuse these actors in other workflows and contexts. In general, *ad hoc* approaches for managing nested collections in scientific workflows leads to code duplication and tightly couples actor implementations with workflow designs; hampers rapid prototyping of workflows and associated data structures; makes comprehension, reuse, and refactoring of existing workflows difficult; and limits reuse of actors designed for these workflows.

Our solution is to provide explicit support for developing "collection-aware" actors. These actors employ a common framework for managing nested collections efficiently and transparently. Moreover, workflows composed from collection-aware actors do not suffer from the reuse limitations inherent in *ad hoc* approaches to managing nested collections. The lower panel of Figure 2 shows collection-oriented workflows equivalent to the conventional workflows in the upper panel. Note that introducing additional levels of data nesting does not change the collection-oriented workflow definitions. Collection-oriented workflows and actors are by design immune to such changes and thus far more reusable. In this example, each collection-aware actor defines their input of interest using a *scope expression* (*e.g.*, α for A and β for B). The framework automatically performs the necessary control-flow functions for providing each actor with their data of interest. In addition, input data outside of an actor's scope is automatically forwarded downstream.

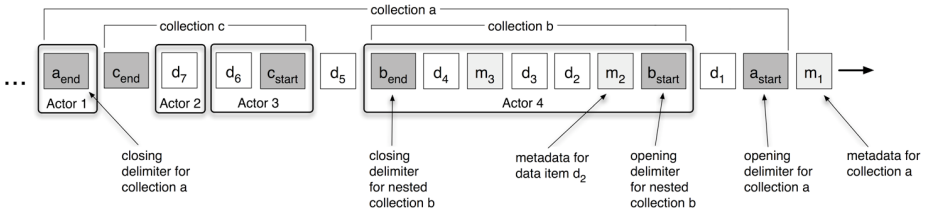

Fig. 3. Collection-oriented workflows represent nested data collections as flat token streams, where collection-aware actors can concurrently process collections

Figure 3 illustrates our approach for streaming nested collections through workflows. Data streams are "flattened" into a sequence of tokens by denoting nested collections via pairs of explicit opening and closing delimiter tokens. Delimited collections may contain data tokens (labeled d_i in Figure 3), explicit metadata tokens (labeled m_j in Figure 3), and other sub-collections (denoted using embedded control tokens, *e.g.*, b_{start} and b_{end}). Metadata tokens are used to carry information that applies to the collections or data items that follow them in the stream. As shown in Figure 3, a series of actors may operate concurrently on the contents of collections. For example, in Figure 3, Actors 1-4 all simultaneously process parts of collection a, Actors 2 and 3 each simultaneously process a part of collection c, and so on.

Figure 4 shows a collection-oriented workflow implemented within Kepler for inferring phylogenetic trees. The AddData actor is used to specify a list of files containing input data in the Nexus file format [14]. The ReadFile actor reads these Nexus files from disk and outputs a generic TextFile collection for each; ParseNexus transforms these text collections into corresponding Nexus collections. The PaupHSearch actor executes the PAUP* [21] external application (as a separate system process) on each Nexus collection it receives, adding the phylogenetic trees it infers to the collection. The PhylipConsense actor applies the CONSENSE[7] external application to the trees inferred by the PaupHSearch actor, adding a consensus tree (reflecting commonalities in the trees inferred by PAUP*) to each Nexus collection. The ExceptionCatcher actor discards Nexus collections that triggered exceptions in upstream actors. The TreeReporter actor displays each tree and associated statistics for each tree in a web-browser interface. Finally, the ComposeNexus and WriteFile actors save the results of analyzing each Nexus collection back to disk in the Nexus file format.

Note that each Nexus collection created by the ParseNexus actor pass through five downstream actors. These actors operate on the Nexus collections in turn, assembly-line style, reading data from the collections, and adding new information back to the collections. In particular, PaupHSearch expects to find data representing a character matrix in each Nexus collection, and PhylipConsense expects to find the phylogenetic trees inferred by PaupHSearch. The TreeReporter actor requires access to both the character matrix and the trees. As described in detail in the next section, each actor in a collection-oriented workflow declares what collection types (*e.g.*, *Nexus*) and data

[7] http://evolution.gs.washington.edu/phylip.html

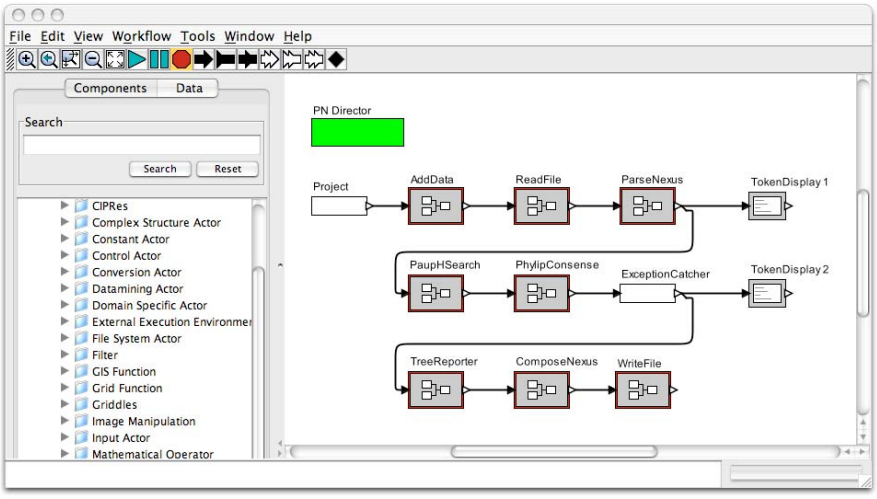

Fig. 4. A KEPLER collection-oriented workflow for inferring phylogenetic trees

types (*e.g.*, *CharacterMatrix*) it operates on using a scope expression. As previously mentioned, the framework transparently passes any data not required by an actor to downstream actors, *i.e.*, an actor is never made aware of data it does not declare interest in. The result is that composing collection-oriented workflows simply entails stringing together actors in an intuitive order (*e.g.*, it makes sense to run TreeReporter after PaupHSearch and PhylipConsense), without regard to the details of the data structures flowing between actors at runtime.

3 Abstract Data Model for Collection-Oriented Workflows

In this section we describe an abstract data model and syntax for representing collection-oriented structures (instances and schemas). Our model represents nested data collections as node-labeled ordered trees that are "flattened" into sequences of underlying data tokens.

3.1 Collection Instances

A *collection instance* in our abstract data model denotes a node-labeled ordered tree (similar to XML). Tree order represents the serialization order of a collection. In general, the order of items within a collection may or may not be "scientifically" meaningful. Node labels are applied to collections, metadata, and data values. Syntactically, a collection is denoted $l[\ldots]$, a metadata value is denoted $@l{:}d$, and a data value is denoted $l{:}d$, where l is a label and d a data value. A data (or metadata) value is either an atomic value such as a string or int, or a complex value represented by an object identifier.

A *collection-oriented sequence* can consist of labeled collections, labeled metadata values, and labeled data values. We require each label within a particular metadata

sequence to be unique. The abstract syntax for sequences is defined by the following grammar. Note that in the abstract syntax, a collection defines a tree by encapsulating a collection-oriented sequence, where each item represents a child of the collection.

$$\begin{array}{lll} s & ::= & v \mid v, s \qquad\qquad (\textit{Sequence}) \\ v & ::= & l{:}d \mid @l{:}d \mid l[s] \qquad (\textit{Data, Metadata, or Collection Value}) \end{array}$$

We convert collection-oriented sequences into KEPLER *token sequences* as follows. Each nested data collection is represented as a flat sequence of tokens within KEPLER(see Figure 3), such that each collection instance is enclosed by special opening and closing delimiter tokens (representing the '[' and ']' collection symbols). Delimiter tokens carry the label of the associated collection. Tokens are also used to store metadata and data items, and to provide actors with explicit access to item labels and to atomic and object-based values.

Nested data collections are often used to model the physical structure of a system under study. The following example, taken from structural biology, represents a portion of a protein structure described in a Protein Data Bank[8] (PDB) file. The PDB collection contains a protein-chain collection that in turn contains two atom objects o_1 and o_2.

$$\text{PDBCollection[ProteinChain[Atom:}o_1\text{, Atom:}o_2\text{]].}$$

The next example defines a Nexus collection nested within a project collection, along with associated metadata.

$$\begin{array}{l} \text{Project[@FilePath:'/myproject/aquatic/turtles.nex',} \\ \qquad \text{Nexus[CharacterMatrix:}o_1\text{, @CI:0.88, Tree:}o_2\text{, @CI:0.82, Tree:}o_3\text{]]} \end{array}$$

This Nexus collection has a file-path metadata value, and each tree within the collection has a CI (consistency index) metadata value. Note that the character matrix and trees inherit the file-path metadata value of the Nexus collection.

In the abstract model, we require metadata values for a given data or collection item to directly precede the item in a sequence. This restriction guarantees that as a data item is received by an actor, the actor has seen the item's associated metadata values. Metadata values are automatically cached for an actor in the KEPLER implementation of collection-oriented workflows. In general, this approach simplifies the processing of metadata, and for many cases limits the amount of data that must be cached, maximizing the performance of pipelining.

The function *descendents*(c) returns the contents of a collection c as a sequence of items, given by a top-down, left-to-right traversal of c. The function *metadata*(v) returns, as a sequence, the metadata values directly associated with a data or collection item v. Metadata values "cascade" to the descendents of a collection, unless otherwise overridden by an item. Thus, the function *metadata**(v) returns all metadata values, as a sequence, for data and collection items v.

The abstract data model for nested data collections is similar to XML. In particular, data and collection items correspond to XML elements, where data "elements" contain only simple content, collection "elements" contain complex content (*i.e.*, subelements), and metadata items correspond to attributes. Our model is simpler in that it

[8] http://www.rcsb.org

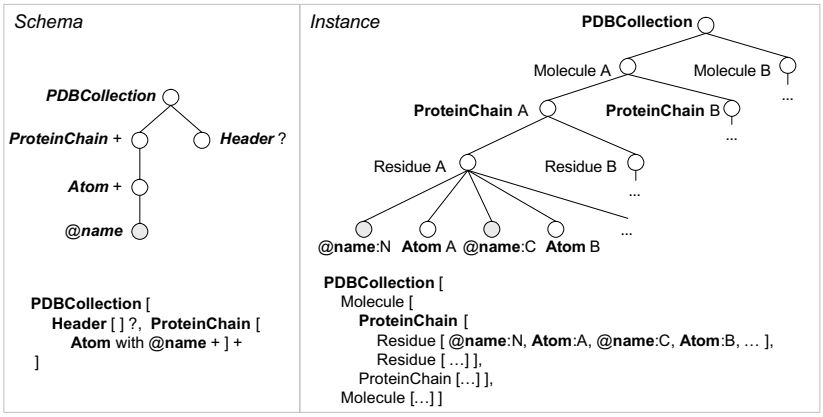

Fig. 5. A collection schema (left) shown as both a tree pattern and using the abstract schema language, and a conforming instance (right) shown both as a nested collection and using the abstract collection language

does not have constructs corresponding to XML documents, identifiers (IDs), references (IDREFs), or mixed content. Also, we treat nesting explicitly as denoting "part-of" relationships, with the result that metadata is inherited by contained "parts." Because of the similarity to XML, we can use standard XML languages over nested collections such as XPath expressions, *e.g.*, to retrieve portions of collection-oriented sequences.[9]

3.2 Collection Schemas

Collection schemas are similar to regular tree grammars [19]. However, our approach is tailored to collection-oriented workflows, in that: (1) we do not assume a "closed" schema model by default, and instead allow conforming instances to contain additional information; (2) we do not restrict the particular nesting levels of sub-collections, and allow conforming instances to contain unspecified intermediate collections; and (3) we do not restrict the ordering of sub-items (collections or data items).

A simple example of a collection schema and conforming instance are given in Figure 5. The schema, shown on the left, defines a PDB collection of interest as containing an optional header collection and one or more protein chain collections, where each protein chain contains one or more atoms having a *name* metadata value. A conforming instance of the schema is shown on the right of Figure 5. The PDB collection instance does not directly contain a protein chain, and instead contains multiple "molecule" collections. Similarly, each protein chain does not directly contain an atom data item, and instead the atoms are nested within residue collections. Thus, unlike with XML Schema or XML DTDs, collection schemas allow instances to have additional items including intermediate collections (*e.g.*, matching PDBCollection//ProteinChain//Atom instead of PDBCollection/ProteinChain/Atom).

A *sequence type* specifies the kinds of items expected within a given sequence. In addition to expected item types, one can also specify item types that are not permissible

[9] With the caveat that metadata values, treated as attributes, "cascade" to nested items.

$$\frac{v :: \tau_v,\ (\neg\exists v' \in s)\ v' :: \tau_v}{v,\ s :: \tau_v} \qquad \frac{v :: \tau_v,\ (\neg\exists v' \in s)\ v' :: \tau_v}{v,\ s :: \tau_v?,\ s :: \tau_v?} \qquad \frac{v :: \tau_v}{v,\ s :: \tau_v+}$$

$$\frac{}{s :: \tau_v*} \qquad \frac{\neg\ (s :: \tau_v)}{s :: \text{not } \tau_v} \qquad \frac{(\forall \tau \in \tau_s)\ s :: \tau}{s :: \tau_s}$$

Fig. 6. Typing rules for occurrence definitions and sequences

in conforming sequences (via the not expression as shown below). An item type is either a data type or a collection type (itself a sequence type). Data and collection types can have occurrence qualifiers restricting the number of times an item may occur within a sequence. The occurrence qualifiers are zero or one (?), one or more (+), zero or more (*), or exactly one (the default). Data and collection types also can have associated metadata types.

A collection type can specify a label and a sequence type. A data or metadata type can specify a label and a value type. Value types (denoted ω) are given by their type names. We do not further specify value structures for complex objects. As for collection instances, metadata types given for a data or collection type "cascade" to nested items.

τ_s	::=	τ_q | not τ_v | τ_s, τ_s	(*SequenceType*)
τ_q	::=	τ_v { + | * | ? }	(*QualifiedType*)
τ_v	::=	τ_d { with m } | τ_c { with m }	(*ItemType*)
τ_d	::=	data | l | :ω | l:ω	(*DataType*)
τ_c	::=	{ l } [{ τ_s }]	(*CollectionType*)
m	::=	τ_m | m, m	(*MetadataSet*)
τ_m	::=	@ { l } { :ω }	(*MetadataType*)

Given a sequence s and a sequence type τ_s, we write $s :: \tau_s$ if s conforms to the type τ_s. Figure 6 defines the typing rules for occurrence definitions and general sequences. Note that the zero-or-many occurrence qualifier, as shown, does not restrict collection contents. However, this qualifier is useful for defining collection-oriented actors, which we discuss in more detail in the next section. The last rule of Figure 6 defines the general case for matching entire sequences. Figure 7 gives the typing rules for data items, collections, and metadata items.

Using schema expressions, it is possible to define standard representations for use in collection-oriented workflows. In particular, a given schema description can be "published," allowing it to be reused by actors. These published schemas also can enable certain forms of static type checking, *i.e.*, to ensure that a given collection instance satisfies the target schema within a workflow. Schema expressions also form the basis for scope expressions, as described in the next section.

4 Scope Expressions for Collection-Oriented Actors

Collection-oriented actors are typically designed to process data within a particular *scope*, as opposed to entire streams of heterogeneous data collections. Here we introduce scope parameters for explicitly defining the portion of an incoming data stream

$$\frac{}{l{:}d :: \mathsf{data}} \qquad \frac{}{l{:}d :: l} \qquad \frac{d :: \omega}{l{:}d :: {:}\omega} \qquad \frac{d :: \omega}{l{:}d :: l{:}\omega}$$

$$\frac{}{l[s] :: [\,]} \qquad \frac{}{l[s] :: l[\,]} \qquad \frac{\text{decendents}(l[s]) :: \tau_s}{l[s] :: [\tau_s]} \qquad \frac{\text{decendents}(l[s]) :: \tau_s}{l[s] :: l[\tau_s]}$$

$$\frac{}{@l{:}d :: @} \qquad \frac{}{@l{:}d :: @l} \qquad \frac{d :: \omega}{@l{:}d :: @\,{:}\omega} \qquad \frac{d :: \omega}{@l{:}d :: @l{:}\omega}$$

$$\frac{\upsilon :: \tau_q,\ @l{:}d' \notin \text{metadata}(\upsilon)}{@l{:}d,\ \upsilon :: \tau_q} \qquad \frac{\upsilon :: \tau_q,\ \upsilon' \in \text{metadata}^*(\upsilon),\ \upsilon' :: \tau_m}{\upsilon',\ \upsilon :: \tau_\upsilon \text{ with } \tau_m}$$

Fig. 7. Typing rules for data, collection, and metadata items

that is relevant to a collection-aware actor. Scope parameters can significantly reduce the effort of developing collection-oriented actors. For example, all data that falls outside of an actor's scope specification can be automatically "passed through" the actor unchanged. The use of scope parameters in this way also facilitates actor reusability, allowing actors to be used on selected portions of complex data streams, and without the actors needing to understand the structure or contents of the entire stream. Workflow designers also can more readily configure a collection-aware actor to work over particular subsets of data by specializing scope parameters, allowing actors to be flexibly reused in distinct workflows. We have found the following types of scope parameters to be useful in practice.

- **Read Scope.** A read scope specifies the portion of an incoming data sequence that is relevant to an actor. Typically, the read scope is used to identify the items generally required for an actor to execute. For example, consider an actor *A* whose read scope is given as a Nexus collection. Here, each particular Nexus collection within an input stream "triggers" *A* to execute.
- **Write Scope.** A write scope specifies where output data is placed within a given stream. For example, actor *A* may add new data items within each input Nexus collection. Alternatively, the actor may add a new collection as a sibling of the Nexus collection, or even replace the Nexus collection with an altogether new type of collection.
- **Iteration Scope.** An iteration scope extends a read scope and describes in more detail (1) what specific data items within the read scope are used by an actor for processing, and (2) how the actor should be invoked over those data items. For example, using an iteration-scope parameter, actor *A* may state that it should be invoked once for each phylogenetic tree in a collection. Alternatively, the actor may state that it should be invoked once over all trees within a collection.
- **Scope Filter.** A scope filter further specializes a read scope. Scope filters are typically used by workflow developers to control processing within a scientific workflow. For example, one might specialize the read scope of actor *A* by adding a metadata restriction (*i.e.*, that a particular metadata value is required) or by requiring the Nexus collection to be nested within another type of collection (*e.g.*, a particular kind of sub-project collection).

Here we focus on read and iteration scope parameters. Our approach is to use collection schemas for expressing read scopes (*i.e.*, for stating the type of incoming data of interest), and to model iteration scopes as queries over schema instances. The result obtained from applying an iteration-scope query to a read-scope instance is then used to control the iteration of the actor (for the particular read-scope instance). We give a simple query language for specifying iteration scopes, where parts of the read scope of an actor are embedded with variable bindings. Both read and iteration scopes are used to facilitate the construction of collection-aware composite actors that wrap traditional actors and subworkflows, as we discuss further in the following section.

The following is an example of a read scope for the PaupHSearch actor of Figure 4.

```
PaupHSearch.read-scope := Nexus[ CharacterMatrix, WeightVector ? ]
```

The PaupHSearch works over Nexus collections that contain exactly one character matrix data item and zero-or-one weight vector. The iteration scope of the PaupHSearch actor is straightforward. For each Nexus collection, the actor consumes the character matrix and weight vector (if it exists), and produces a set of phylogenetic trees. The PaupHSearch iteration scope is given by the following expression.

```
PaupHSearch.iteration-scope ($c, $v) :=
    Nexus[ CharacterMatrix {$c}, WeightVector {$v} ].
```

This iteration expression is shorthand for the following Datalog query.

```
R(c, v) :- Collection(n), Label(n, Nexus), Descendents(n, c),
           Label(c, CharacterMatrix), Descendents(n, v), Label(v, WeightVector).
```

The relations used in the body of the query access portions of a given instance of the read-scope schema. For example, the Label relation associates a collection, data, or metadata item with its label, the Collection relation contains the collection items within the instance, and the Descendents relation relates collection items with their (transitively) contained items.

The read scope of the TreeReporter actor of Figure 4 is given by the following expression.

```
TreeReporter.read-scope := Nexus[ CharacterMatrix, Tree + ].
```

In this case, the TreeReporter actor displays a report for each tree in the nexus collection using the given character matrix. Thus, for a given nexus collection, the actor is *repeatedly* invoked, once for each tree. This invocation pattern is expressed by the following scope iteration.

```
TreeReporter.iteration-scope ($c, $t) :=  Nexus[ CharacterMatrix {$c}, Tree {$t} ].
```

Finally, the read scope of the ComposeNexus actor of Figure 4 is given by the following expression.

```
ComposeNexus.read-scope :=  Nexus[ CharacterMatrix ?, WeightVector ?, Tree *]
```

The ComposeNexus actor converts an optional character matrix, weight vector, and a list of zero-or-more trees into a Nexus file. Note here that the actor is invoked exactly once for each input Nexus collection, unlike the TreeReporter actor, which is invoked once per tree. This invocation pattern is described by the following iteration scope.

```
ComposeNexus.iteration-scope ($c, $v, collect($t in $n)) :=
    Nexus{$n}[ CharacterMatrix {$c},  WeightVector {$v}, Tree {$t} ]
```

The collect expression constructs a list of trees, where each tree is contained in the given Nexus collection. Every collect expression in an iteration scope consists of a data or metadata variable (in this case $t) combined with a collection variable (in this case $n).

In general, an iteration scope defines a mapping from instances I of a collection schema S to a relation $R(x_1, \ldots, x_n)$, for $n \geq 1$. We call each x_i of R an *attribute* of the iteration scope. Let I be an instance of the read-scope S. We write $R(I)$ to denote the result of applying the iteration scope to I, where each x_i attribute value for a tuple in $R(I)$ consists of either a metadata value, a data value, or a list of values resulting from a collect expression. Further, the actor is invoked once for each tuple in $R(I)$. We note that $R(I)$ can be "lazily" constructed (similar to a standard database iterator) such that the actor is invoked immediately as each new tuple is obtained.

5 Developing Collection-Aware Actors

We provide two approaches for developing collection-aware actors in Kepler. The first, which we discuss in more detail in [17], is to directly implement collection-aware actors *natively* using a Java API. This API simplifies the implementation of collection-aware actors by providing comprehensive support for streaming, managing, and operating on nested data collections. However, we do not expect all actors to be developed in this way. A large number of "legacy" conventional actors already exist and are in use, including web-services and application components that are not designed to be collection aware. Furthermore, it is often easier and more intuitive to implement conventional Kepler actors, especially those actors that do not explicitly operate on collections. Examples include straightforward data-transformation actors that take a single input and produce a single output, and actors that provide low-level functions for reading and writing files.

Thus, the second approach for developing collection-aware actors, which we introduce here, involves wrapping traditional actors, or entire subworkflows, within *collection-aware composite actors*. This approach facilitates the use of Kepler itself for specifying collection-aware actors, allows conventional actors to be reused within multiple collection-aware actors, and reduces the need for writing *ad hoc*, single-purpose collection-aware actors from scratch.

To demonstrate the approach, Figure 8 shows how the PaupHSearch, TreeReporter, and ComposeNexus composite actors of Figure 4 are defined. Each composite actor contains a subworkflow employing an SDF director and one or more conventional actors. The ComposeNexus subworkflow illustrates how a single conventional actor may be wrapped in a composite to yield a collection-aware version of the actor. The

subworkflow input ports labeled CharacterMatrix?, WeightVector? and Tree* map to attributes of the iteration scope parameter of the enclosing collection-aware composite actor. Like any other collection-oriented actor, the ComposeNexus subworkflow is invoked each time a match is found for the iteration scope of the actor. On each invocation, data values of the iteration-scope attributes are passed to the corresponding subworkflow input ports, the actor within the subworkflow operates on these data, and the outputs written by the enclosed actor are accumulated by the subworkflow output port labeled String. The enclosing composite actor then inserts the output of the subworkflow back into the data stream. The labels on the ports specify the types and quantity of data consumed or produced by the subworkflow, and provide anchors for mapping the iteration scope attributes to the ports.

The PaupHSearch and TreeReporter composite actors are more sophisticated. Both create and destroy temporary file system directories for running external processes, on each invocation, by employing the CreateProcessEnvironment and DestroyProcessEnvironment conventional actors. Both run external applications (PAUP* and DRAWGRAM[10]), write temporary files for these external programs to read, and parse output files created by these programs. Note that the ComposeNexus conventional actor is used both in PaupHSearch and in the ComposeNexus composite actor.

Employing the SDF director in collection-aware composite actors, rather than the PN director used to control the overall collection-oriented workflow, offers a number of practical advantages. The use of SDF simplifies the specification of these subworkflows, requiring each actor to have a well-defined token consumption and production rate. In addition, these lower-level actors can benefit from the optimized static schedule computed by the SDF director, since they typically perform a single function or are meant to be executed only once per composite invocation. The ability to use multiple models of computation in a single overall workflow is one of the main strengths of KEPLER, and is essential for supporting these SDF-based subworkflows in our collection-oriented workflow framework.

6 Conclusion

Our collection-oriented framework shares a number of similarities to XML-based approaches. For example, the way in which collection-aware actors operate on pipelined nested collections has similarities with some XML stream processing techniques [6]. The approach is also similar in spirit to list processing constructs in functional programming [2] as well as dataflow programming [18]. Because the abstract model of nested data collections is (essentially) a subset of XML, we can leverage and adapt existing XML-based query processing [7] and optimization techniques [8] for managing nested collections. For example, algorithms for XML-based publish and subscribe architectures [23] are relevant for applying actor scope parameters to incoming data streams, and iteration-scope expressions can leverage work in XML query optimization and on languages such as XPathLog [16].

Scientific workflows play an important role in a number of ongoing large research projects dealing with scientific data management, and represent an emerging paradigm

[10] http://evolution.gs.washington.edu/phylip.html

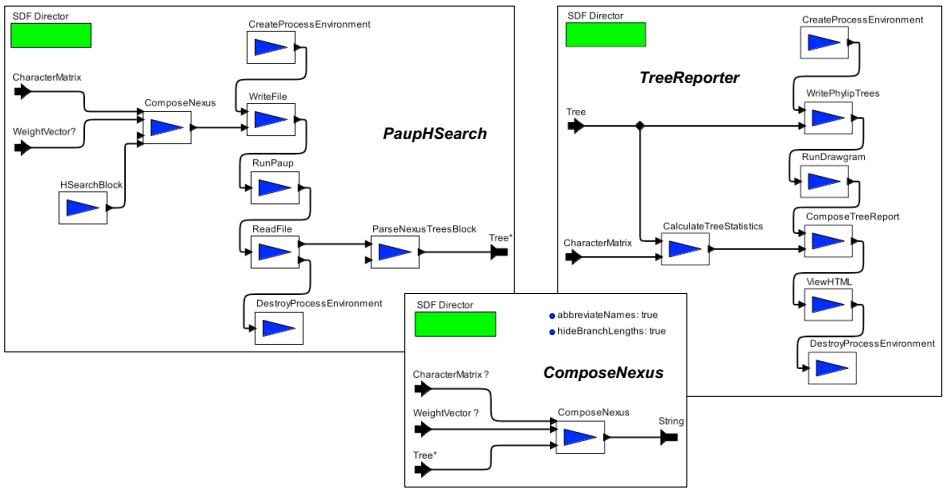

Fig. 8. The PaupHSearch, TreeReporter, and ComposeNexus collection-aware actors defined in terms of conventional actors

for analyzing and integrating biological data from diverse sources. The development of "rigid" workflow modeling and design frameworks has recently been identified as a major bottleneck for scientific workflow reuse and repurposing [5]. We have found that this lack of flexibility is often due to the use of control-flow within workflows for managing, integrating, and analyzing inherently complex life-science data. The collection-oriented framework extends the capabilities of existing systems by facilitating the management of scientific data within scientific workflows. In particular, collection-oriented workflows are often significantly simpler and more intuitive than their conventional counterparts, can support higher-levels of concurrency and pipelining, and allow flexible actor configuration enabling greater levels of actor reuse. By additionally allowing collection-aware actors to be composed from conventional actors and KEPLER sub-workflows, our approach can support the reuse and repurposing of a wide variety of actors and workflows.

References

1. I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludäscher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *SSDBM*, 2004.
2. P. Buneman, S. A. Naqvi, V. Tannen, and L. Wong. Principles of Programming with Complex Objects and Collection Types. *Theoretical Computer Science*, 149(1), 1995.
3. S. Davidson, C. Hara, and L. Popa. Querying an Object-Oriented Database using CPL. In *Brazilian Symposium on Databases (SBBD)*, 1997.
4. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping Scientific Workflows onto the Grid. In *European Across Grids Conference*, 2004.
5. A. Goderis, C. Goble, U. Sattler, and P. Lord. Seven Bottlenecks to Workflow Reuse and Repurposing. In *ISWC*, 2005.

6. L. Golab and M. T. Özsu. Issues in Data Stream Management. *ACM SIGMOD Record*, 2003.
7. A. K. Gupta and D. Suciu. Stream Processing of XPath Queries with Predicates. In *ACM SIGMOD*, pages 419–430, 2003.
8. Z. G. Ives, A. Y. Halevy, and D. S. Weld. An XML Query Engine for Network-Bound Data. *VLDB Journal*, 11(4):380–402, 2002.
9. G. Kahn and D. B. MacQueen. Coroutines and Networks of Parallel Processes. In *IFIP Congress*, 1977.
10. E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Trans. Comput.*, C-36, 1987.
11. U. Leser and F. Naumann. (Almost) Hands-Off Information Integration for the Life Sciences. In *Conference on Innovative Data Systems Research (CIDR)*, 2005.
12. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience*, 2005.
13. R. S. MacLeod, D. M. Weinstein, J. Davison de St. Germain, C. R. Johnson, S. G. Parker, and D. Brooks. SCIRun/BioPSE: Integrated Problem Solving Environment for Bioelectric Field Problems and Visualization. In *Symposium on Biomedical Imaging (ISBI): From Nano to Macro*, 2004.
14. D. Maddison, D. Swofford, and W. Maddison. NEXUS: An Extensible File Format for Systematic Information. *Systematic Biology*, 46(4):590–621, 1997.
15. S. Majithia, M. S. Shields, I. J. Taylor, and I. Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. In *ICWS*, 2004.
16. W. May. XPath-Logic and XPathLog: A Logic-Programming-Style XML Data Manipulation Language. *Theory and Practice of Logic Programming*, 4(3):239–287, 2004.
17. T. McPhillips and S. Bowers. An Approach for Pipelining Nested Collections in Scientific Workflows. *ACM SIGMOD Record*, 34(3):12–17, 2005.
18. J. Morrison. *Flow-Based Programming*. Van Nostrand Reinhold, 1994.
19. M. Murata, D. Lee, and M. Mani. Taxonomy of XML Schema Languages using Formal Language Theory. In *Extreme Markup Languages Conferences*, 2001.
20. T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: A Tool for the Composition and Enactment of Bioinformatics Workflows. *Bioinformatics*, 20(17), 2004.
21. D. Swofford. PAUP*: Phylogenetic Analysis Under Parsimony (*and Other Methods). Version 4. Sinauer Associates, Sunderland, Massachusetts.
22. D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency – Practice and Experience*, 17(2-4), 2005.
23. F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, and J. Myllymaki. Implementing a Scalable XML Publish/Subscribe System Using a Relational Database System. In *ACM SIGMOD*, pages 479–490, 2004.