An Approach for Pipelining Nested Collections in Scientific Workflows

Timothy M. McPhillips
Natural Diversity Discovery Project
tmcphillips@nddp.org

Shawn Bowers[*]
UC Davis Genome Center
sbowers@ucdavis.edu

ABSTRACT

We describe an approach for pipelining nested data collections in scientific workflows. Our approach logically delimits arbitrarily nested collections of data tokens using special, paired control tokens inserted into token streams, and provides workflow components with high-level operations for managing these collections. Our framework provides new capabilities for: (1) concurrent operation on collections; (2) on-the-fly customization of workflow component behavior; (3) improved handling of exceptions and faults; and (4) transparent passing of provenance and metadata within token streams. We demonstrate our approach using a workflow for inferring phylogenetic trees. We also describe future extensions to support richer typing mechanisms for facilitating sharing and reuse of workflow components between disciplines. This work represents a step towards our larger goal of exploiting collection-oriented dataflow programming as a new paradigm for scientific workflow systems, an approach we believe will significantly reduce the complexity of creating and reusing workflows and workflow components.

1. INTRODUCTION

New instrumentation, automation, computers, and networks are catalyzing high-throughput scientific research. These technologies promise to deliver data at rates orders of magnitude greater than in the past. Amid high expectations, however, is a growing awareness that existing software infrastructure for supporting data-intensive research will not meet future needs. Researchers in high-energy physics, biology, nanotechnology, climate research, and other disciplines recently reported at the 2004 Data-Management Workshops [7] that current technologies for managing large-scale scientific research do not satisfy even current needs and warned of a "coming tsunami of scientific data." They identified critical computing challenges including: (1) integrating large data sets from diverse sources; (2) capturing data provenance and other metadata; and (3) streaming data through geographically distributed experimental and computing resources in real time. An emerging challenge is the need to make high-throughput automation technologies, developed and made cost-effective by large research consortia, available to medium-sized collaborations, small research groups, and individual investigators through virtual laboratories composed of network-accessible research tools. Effective information-intensive research by small groups requires automated workflow approaches where computing infrastructure integrates disparate resources and explicitly manages project data [7].

The goals of the *Natural Diversity Discovery Project* (NDDP) [19] illuminate the challenges of supporting scientific workflows for genomics and bioinformatics. Aiming to help the public understand scientific explanations for the diversity of life, the NDDP is developing a virtual laboratory for inferring and analyzing evolutionary relationships between organisms, i.e., phylogenetic trees [8]. Professional research tools and a web-based discovery environment will enable evolutionary biologists and the general public to: (1) infer, display, and compare phylogenetic trees based on morphology, molecular sequences, and genome features; (2) correlate these phylogenies with events in Earth history using molecular clocks and the fossil record; (3) iterate over alternative phylogenetics methods, character weightings, and algorithm parameter values; (4) maintain associations between phylogenies and the data, methods, parameters, and assumptions used to infer them; (5) share workflows and results; and (6) repeat studies reported by others and note the effects of varying data sets, approaches, and parameters.

To support these research and discovery environments, the NDDP is addressing the following requirements for scientific workflows:

- **Workflows must support operations on nested collections of data.** Data sets for phylogenetics, and bioinformatics in general, can be large, complex, and nested in structure [10]. Workflows must operate efficiently on these sometimes deeply nested collections of data, while maintaining the associations they signify.

- **Workflow results must be repeatable.** Researchers need to be able to repeat the work of others easily and reliably. Workflow infrastructure must automatically record how results were obtained and allow others to use this information to reproduce the results. The provenance of input data and important intermediate results must be associated automatically with outputs.

- **Workflow definitions must be reusable.** Researchers must be able to develop generic, reusable workflows from existing workflow components. Moreover, users of the NDDP web-based discovery environment must be able to apply predefined workflows to data sets of their choice. Running a new set of data through a previously defined workflow must not entail manual reconfiguration of component parameters or interconnections.

- **Workflows and components must be robust.** Exceptions thrown for particular data or parameter sets must not disrupt operations on unrelated sets. Similarly, it should be possible for a workflow author to specify the consequences of faults during workflow processing.

[*]Supported in part through NSF/ITR 0225676 (SEEK).

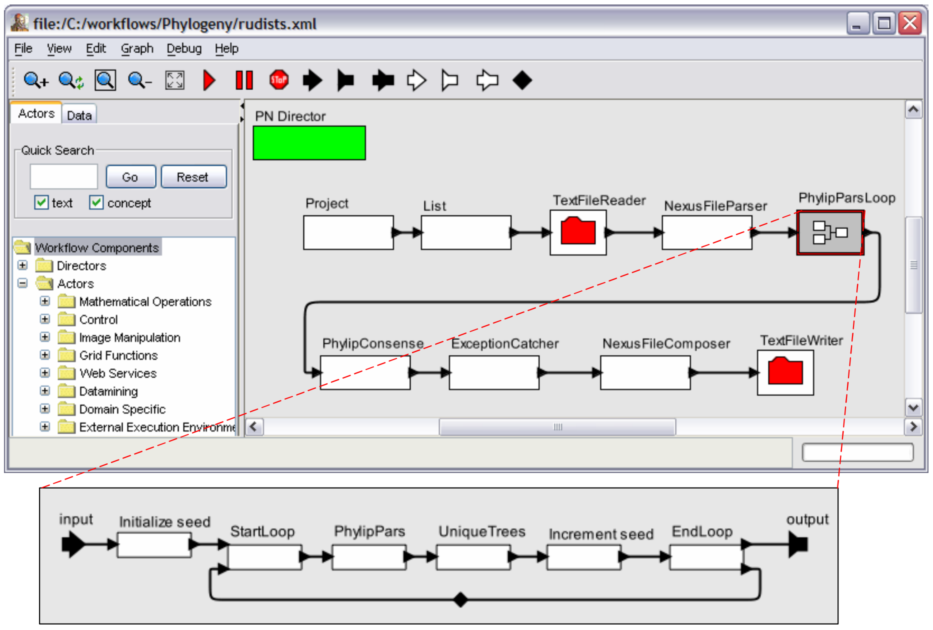

Figure 1: A collection-aware KEPLER workflow for inferring phylogenetic trees. The PhylipParsLoop *composite* actor (top) contains a nested *sub-workflow* (bottom), which iteratively executes the PARS algorithm.

We report our experiences addressing these requirements using the KEPLER scientific workflow system and detail the resulting approach. We give a brief description of KEPLER in Section 2 and describe our extensions for supporting pipelined nested data collections in scientific workflows in Section 3. Our approach treats pipelined dataflow as sequences of data tokens containing special, paired control tokens that delimit arbitrarily nested collections. We show how our implementation of these capabilities within KEPLER automates the management of nested collections and simplifies the development of "collection-aware" pipelined components. In Section 4 we describe planned extensions for supporting rich data typing of collections and workflow components. These extensions will further support the design and development of pipelined workflows and will facilitate mechanisms for workflow verification and analysis. Related work is discussed in Section 5.

2. THE KEPLER SYSTEM

KEPLER [12] is a Java-based, open-source scientific workflow system being developed jointly by a collaboration of application-oriented scientific research projects. KEPLER extends the PTOLEMY II[1] system (hereafter, PTOLEMY) with new features and components for scientific workflow design and for efficient workflow execution using distributed computational and experimental resources [16]. PTOLEMY was originally developed by the electrical engineering community as a visual dataflow programming application [14] that facilitates *actor-oriented programming* [13]. In PTOLEMY and thus in KEPLER, users develop workflows by selecting appropriate components (called *actors* or *blocks*) 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 1. Actors have *input ports* and *output ports* that provide the communication interface to other actors. Actors can be hierarchically nested, using *composite actors* to contain sub-workflows. Control-flow elements such as branches and loops are also supported (see the bottom of Figure 1). 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[2] and Matlab, database queries, SRB[3] commands, and so on.

In PTOLEMY, dataflow streams consist of data *tokens*, which are passed from one actor to another via actor connections. PTOLEMY differs from other similar systems (including those for scientific workflows) 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 requiring different models of computation. PTOLEMY includes directors that specify, e.g., process network, continuous time, discrete event, and finite state computation models.

As an example, the *Process Network* (PN) director executes each actor in a workflow as a separate process (or thread). Connections (or *channels*) are used to send (i.e., stream) sequences of data tokens between actors, and actors map input sequences to output sequences. Actors communicate asynchronously in process networks through buffered channels implemented as queues of effectively unbounded size. Thus, the PN director can be used to pipeline data tokens through scientific workflows, enabling highly concurrent execution.

3. PIPELINING NESTED DATA COLLECTIONS

As previously noted, scientific data sets are often large. Operating efficiently on such data sets can require pipelined processing of data set contents, a task for which scientific workflows are

[1]http://ptolemy.eecs.berkeley.edu/index.htm

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

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

potentially well suited. PTOLEMY, however, does not provide explicit support for pipelining the *nested* structures typical of scientific data. PTOLEMY represents collections as individual tokens, meaning that pipelining, e.g., using process networks, occurs at the granularity of an entire collection. This approach precludes efficiency gains that might be realized by operating on collection members *concurrently* in a pipelined fashion.

To work around this limitation, workflow authors use a variety of approaches. One approach, for example, is to send the members of a collection as individual tokens, and use a separate channel (connection) to send a token count denoting the size of the collection being processed. For nested collections, this approach becomes even more complicated as it requires a large number of additional connections.

Alternatively, an actor can send special "control" tokens mixed in with the data to delimit the beginning and end of a collection (or nested collection). This approach is similar to the use of open-bracket and close-bracket information packets described by Morrison [18] and the approach proposed in Ludäscher et al [15] for supporting "pipelined arrays."[4] However, *ad hoc* use of control tokens leads to code redundancy and tightly couples actor implementation with workflow design. These problems in turn hamper rapid prototyping of workflows and associated data structures; make comprehension, reuse, and refactoring of existing workflows difficult; and limit reuse of actors designed for these workflows.

Our solution is to provide explicit support within KEPLER for pipelining nested data collections using control tokens. Specifically, we have extended KEPLER to denote nested collections using explicit, paired opening and closing delimiter tokens inserted into the data streams. We have also added to KEPLER generic operations for managing these collections.

Figure 2 illustrates how nested data collections can be streamed through consecutive actors in our approach. The collection labeled b is nested within the collection labeled a using explicit start and end control tokens. Delimited collections may contain data tokens (labeled d_i in Figure 2), explicit metadata tokens (labeled m_j in Figure 2), and other sub-collections (denoted using nested control tokens, e.g., b_{start} and b_{end}). Metadata tokens are used to carry information that applies to a collection as a whole, including data provenance. As shown in Figure 2, each actor within the pipeline is able to execute concurrently on a collection's contents. In particular, each actor is processing a part of a and its metadata simultaneously: actor 3 is processing the beginning of the a collection, actor 2 is processing the nested b collection, and actor 1 is processing the end of the a collection.

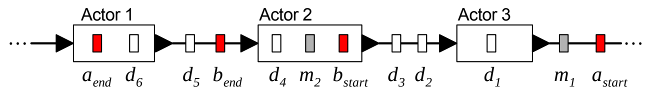

Figure 2: Pipelining data collections via explicit control tokens.

The rest of this section describes the details of our implementation. We first discuss extensions to PTOLEMY for making actors "collection-aware." We describe how an actor can easily "listen" for collections of interest. We discuss the use of collection metadata, support for provenance, and our approach to exception handling. Finally, we describe a real-world, collection-based workflow for inferring phylogenetic trees.

[4]This approach is also similar to the use of XML in stream-based query frameworks.

3.1 Collection-Aware Actors

In PTOLEMY, atomic actors are implemented by extending one of the existing actor classes. Creating a new atomic actor entails overriding the *fire* operation (as well as other related methods), which is called by a director when the actor is "activated." Within fire, an actor may read (consume) tokens from input ports, operate on input data, and write (produce) tokens to output ports. In process networks, for example, an indefinite number of tokens may be received and sent each time fire is called.

We encapsulate the complexity of creating, managing, and processing nested data collections by introducing a new type of actor called *CollectionActor*, and a new system component for managing collections called *CollectionManager*. Figure 3 shows a simplified definition of these classes. Collection-processing actors are derived from CollectionActor. Instances of CollectionManager are used to manipulate particular collections. The CollectionActor base class facilitates collection nesting by maintaining a stack of CollectionManager objects corresponding to all collections concurrently processed by an actor.

Rather than reading tokens directly from input ports, collection actors operate on collections from within methods analogous to SAX API event handlers for parsing XML files. The CollectionActor fire method triggers calls to the *handleCollectionStart* method when the opening delimiter for a collection is received; the *handleData* or *handleMetadata* method when a data or metadata token is received; and the *handleCollectionEnd* method when the closing delimiter for a collection is received. These calls pass the CollectionManager object associated with the incoming collection to these event handlers, and the newly received token to the handleData and handleMetadata methods. This event-based approach to processing collections allows collection-aware actors to be mixed freely with actors that operate on data tokens individually; the latter actors override the handleData method alone, thereby ignoring events related to collection structures and metadata.

Collection actor output is "indirect" as well. An actor may add data or metadata to a collection it is processing using methods provided by the associated CollectionManager object. An actor may create a new collection within another collection and add data or metadata to it; and it may replace data or metadata or copy the information to other collections. Collection actors specify the disposition of incoming collections, data, and metadata via the return values of the event handlers. The return value of the handleCollectionStart method declares whether the actor will further process a collection and whether the collection should be discarded or forwarded to the next actor in the workflow. Similarly, the return values of the handleData and handleMetadata methods indicate whether the token in question should be forwarded or discarded. Incoming information not discarded by an actor is streamed to succeeding actors in the workflow as the collection is received and processed.

3.2 Collection Types and Paths

Each collection is associated with a type (implemented as a Java class) denoting a (conceptual) scientific or data-management resource. For example, a collection with the type *Nexus* contains data or results associated with one or more phylogenetics computations, while a *TextFile* collection contains string tokens representing the contents of a text file.

Collection types simplify the processing of collections. Each instance of a collection actor has a *CollectionPath* parameter that specifies what conceptual types of collections and data the actor can handle. Collections and data tokens with types matching the CollectionPath value trigger collection-handling events, e.g., calls to

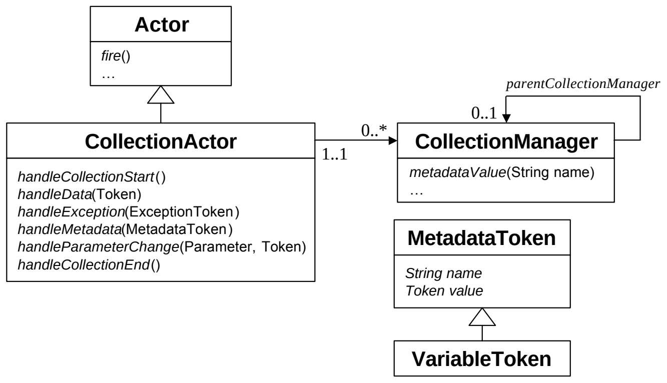

Figure 3: A simplified UML representation of the collection actor and manager classes.

the actor's handleCollectionStart and handleData methods. Collections and data not matching the CollectionPath value are streamed silently to the next actor in the workflow (i.e., no further processing is required by the actor to forward the tokens). Collection paths are expressed using a simplified XPath-style syntax expressed against type names and may specify collection types occurring anywhere in a hierarchy; parent and child collections not matching the path are ignored (although metadata for parent collections are always available). Thus, workflow developers may use the CollectionPath parameter to operate selectively on particular collection and data types, while actor authors may fix the value of these parameters to simplify actor implementation.

3.3 Context-Dependent Operations

Independent data sets passing through a workflow may require different actor behavior within a single workflow run. Actors used in such workflows must be dynamically configurable and able to operate context dependently. Metadata tokens can be used to communicate this context to actors.

An instance of the *MetadataToken* class (see Figure 3) stores the name of a metadata item and an embedded token representing the item's value. Any number of metadata tokens with distinct names may be placed within the sequence of tokens comprising a single collection.

Actors may use metadata values to tune their own behavior appropriately for the current collection. Actors may observe metadata sequentially via the *handleMetadata* method or on demand using the CollectionManager *metadataValue* method after the metadata tokens have been received. The latter random access method traverses the stack of successively enclosing collections to return the first metadata value corresponding to the given name. Thus, the context for actor behavior may be defined at any level within a set of nested collections and overridden at successively lower levels.

In KEPLER, actor *parameters* can be used to specify default actor behavior prior to workflow execution. Passing actor configuration information within collections enhances this capability. In particular, our framework allows instances of *VariableToken*, a subclass of MetadataToken, to automatically override the values of actor parameters at run time. A previous value of the parameter is restored when the end of a collection that overrides the parameter is reached. Among other advantages, variable tokens make it easy for workflow users to apply particular parameter values to subsets of data flowing through dynamically reconfigurable workflows (see the workflow described at the end of this section for an example).

3.4 Provenance Support

Collection metadata provide a convenient mechanism for recording provenance. The origin of input data, intermediate results, and workflow outputs can be described using metadata tokens, and this provenance information can be used to reproduce workflow results later. Inserting metadata directly into the data stream is an effective alternative to storing provenance information within databases or other persistent storage during workflow execution. The approach ensures that metadata is associated with the appropriate collections of data even when independent data sets are processed concurrently by pipelined workflows. It also reduces the burden on the authors of actors that do not require access to metadata: the event-based model for handling data and metadata tokens makes provenance annotations effectively "invisible ink" to actors that do not override the handleMetadata method of CollectionActor. Finally, the in-stream approach to recording provenance is convenient for distributed computing environments where all nodes may not have access to a single shared resource for storing metadata.

3.5 Exception Handling

Exception handling is a significant hurdle to supporting complex scientific workflows [16]. Exceptions can occur for many reasons. For example, many NDDP workflows include actors that wrap existing software. These legacy scientific application programs vary widely in robustness. Inappropriate input data or parameters can result in program faults. Moreover, many scientific applications are prone to crashes even when given valid instructions and data. Without mechanisms for handling exceptions in pipelined workflows, an error caused by a single data set (or collection) can result in the sudden termination of an entire workflow run.

We have addressed these issues by adding support in KEPLER for associating exceptions with collections. A collection-aware actor that catches an external application error (or other exception) may add an ExceptionToken to the collection that caused the error. This actor may then proceed to operate on the next collection. A downstream exception-catching actor can filter out collections that contain exception tokens, and may do so at a level in the collection nesting appropriate for the particular application. This approach limits the effects of exceptions to the collections that trigger them.

3.6 Example: Inferring Phylogenetic Trees

We have used all of the above KEPLER extensions in a number of NDDP workflows. One such workflow for inferring phylogenetic trees is shown in Figure 1. The workflow is run by specifying a list of files containing input data in the Nexus file format [17]. The TextFileReader actor reads these Nexus files from disk and outputs a generic TextFile collection for each; NexusFileParser transforms these text collections into corresponding Nexus collections. PhylipParsLoop is a composite actor containing the sub-workflow shown at the bottom of Figure 1. Within this sub-workflow the PhylipPars actor executes the PARS program (as a separate system process) on each Nexus collection it receives, adding the phylogenetic trees it infers to the collection. The sub-workflow iteratively executes the PARS application using the StartLoop-EndLoop construct.

The actors labeled *Initialize seed* and *Increment seed* are instances of the SetVariable class. A SetVariable actor may add or update the value of a VariableToken using a configurable expression referring to collection metadata or variable values. The first instance of SetVariable adds a VariableToken named *jumbleSeed* to each Nexus collection, while the second increments the value of this variable. The VariableToken overrides the value of the jumbleSeed parameter of the PhylipPars actor, causing the PARS application to jumble the order of analyzed taxa differently on each execution.

The UniqueTrees actor removes redundant trees from the Nexus collection on each pass through the loop. Both PhylipPars and UniqueTrees update the *treeCount* metadata item. The loop is executed until a minimum number of unique trees has been found, or the maximum allowed number of cycles has been performed. The PhylipConsense actor applies the CONSENSE external application to the trees inferred by the PhylipParsLoop sub-workflow, adding a consensus tree (reflecting commonalities in the trees inferred by PARS) to each Nexus collection. The rest of the workflow discards Nexus collections that triggered exceptions and writes out the remaining Nexus collections to disk. The PARS and CONSENSE programs are part of the Phylip phylogeny inference package[5].

The visual simplicity of the workflow (as shown in Figure 1) highlights the power of the pipelined collection approach in KEPLER. Fairly complex, nested data collections (including trees and character matrices) are streamed through the pipeline; the actors PhylipPars and PhylipConsense wrap external scientific applications; and conditional control-flow constructs are used. All this is achieved without introducing numerous connections between actors or customizing actor Java code for the particular workflow. Moreover, the flow of data is pipelined automatically, with workflow components operating concurrently.

An *ad hoc* approach to processing collections would significantly complicate the visual appearance and development of this workflow. Even the relatively straightforward UniqueTrees actor likely would need at least two input ports, one for receiving tokens representing the trees to compare, and another for receiving a token count indicating how many trees the actor should expect to receive. This actor might require two output ports for similar reasons. Other actors in the workflow would require multiple ports and connections as well. Supporting the looping, exception handling, and metadata processing features of this workflow would introduce even more complexity. Passing aggregate tokens (e.g., tokens containing arrays of other tokens) between actors would reduce this complexity somewhat at the cost of limiting concurrent actor execution. In contrast, our delimited-collections approach not only allows multiple, independent collections to stream through workflows at the same time (e.g., if multiple Nexus input files are specified by the List actor in Figure 1), it also allows the *members* of each collection to be processed concurrently in pipelined fashion as appropriate (as illustrated in Figure 2). Most significant, the above alternative implementations would require the scientist developing the workflow to pay considerable attention to the detailed control-flow aspects of the workflow, rather than focusing primarily on the scientific tasks modeled succinctly in Figure 1.

4. TYPING EXTENSIONS

Type safety is a key element of robust programming environments. PTOLEMY provides type safety by enabling actor authors to specify the types of tokens that may pass through actor input and output ports. Exceptions are thrown if incompatible tokens are received or sent by such ports. Further, it is possible to determine prior to workflow execution if connections are type-safe. Such static workflow analysis is desirable both for workflow design and for notifying users of potential problems prior to workflow execution.

In the context of pipelining nested data collections, however, the static typing approach employed by PTOLEMY is no longer appropriate. Any type of token can occur within a stream, including metadata and delimiter token types. Only certain types of collections and data (specified, e.g., using collection paths) are processed by an actor, while all non-relevant data is forwarded transparently to downstream components. Restricting an input port to a particular structural type would unnecessarily limit the applicability of an actor to streams containing information only of that type; and restricting an output port to a particular type would cause unnecessary exceptions for all non-relevant data.

PTOLEMY type checking is disabled in our current implementation, but we intend as future work to "resurrect" the typing of actors as follows. Using a collection-based typing language, similar to content model definitions in XML Schema and DTDs (and subtyping rules, e.g., [11]), we allow actors to explicitly define the types of collections they process both conceptually (see below) and structurally. In this way, collection paths are extended to support structural collection types, expressed as restrictions on the allowable types of data tokens and sub-collections they may contain.

In addition, actor authors can provide output types specifying the types of collections (both conceptually and structurally) an actor can produce upon firing. In a static analogy to the current dynamic use of return types in the *CollectionActor* class, we allow actor authors to specify explicitly whether an input collection of a certain type will be forwarded or discarded by the actor. Actor authors can also specify whether an output collection will result from creating a new collection or by altering particular input collections. With these specifications, KEPLER can perform static type analyses, e.g., to determine that a particular connection will result in an actor never firing, or that an actor may receive a conceptually appropriate collection from an upstream actor, but with an unsupported structural type. For output types, KEPLER can also perform runtime type checking using these specifications.

The current *CollectionPath* implementation defines conceptual types using a hierarchy of Java classes. This approach works when the number of collection types is small, the types are fairly static, and the actors operating on these types are developed by the same organization. As other organizations develop their own collection-based actors, developers independently define their own collection types, and users begin mixing collection actors and types originally developed for different disciplines, more robust and richer typing mechanisms will be needed. As future work we intend to adopt the ontology-based *hybrid typing* approach [2, 3] in KEPLER to specify both structural and conceptual types of collections.

In particular, KEPLER extends the type system of PTOLEMY by separating the concerns of conventional data modeling (structural data types) from conceptual data modeling (semantic data types). A semantic type represents an ontology concept, expressed in description logic (e.g., in OWL-DL).[6] Optionally, hybrid types permit structural types and semantic types to be linked through *semantic annotations*, expressed as logical constraints. Within KEPLER, hybrid types enable concept-based searching for actors, can be used to further propagate and refine known (structural or semantic) types in scientific workflows, and can help to infer (partial) structural mappings between structurally incompatible workflow components.

5. RELATED WORK

A number of scientific workflow systems have recently emerged [1, 6, 20, 22, 21, 4]. To the best of our knowledge, none offer approaches for pipelining and managing nested data collections for workflows consisting of external and opaque (as opposed

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

[6]Thus, different organizations can define their own ontologies (terminologies) and easily articulate mappings among them for interoperability.

to declarative or query-based) components. We believe that KEPLER is well-suited for our extensions because of its advanced and well-defined computation models inherited from PTOLEMY, and because it provides an elegant extension mechanism via actor-oriented design. The way collection actors and managers operate on pipelined nested collections has similarities with some XML stream processing techniques [9], which thus are good candidates for collection processing optimization strategies. Finally, our approach to pipelining workflows is similar in spirit to list processing constructs in functional programming [5] as well as dataflow programming [18]. We believe that many constructs in these approaches may help make scientific-workflow design and development simpler and more intuitive for scientists.

6. CONCLUDING REMARKS

Our approach and associated KEPLER extensions facilitate the concurrent execution of collection-oriented scientific workflows. In particular, our framework automates the pipelining of individual data tokens within nested collections passing through a workflow, and at the same time explicitly maintains the inherent structure of the collections. The approach also allows metadata to be inserted into token streams on the fly; this metadata can be used for recording provenance and dynamically customizing actor behavior. Similarly, the repercussions of external application faults and other exceptions can be limited and controlled through special exception tokens. Our event-based model for handling nested collections makes collection-aware actors simple to implement and maintain, and workflows based on them easy to prototype and extend. Our approach simplifies collection-oriented scientific workflows by eliminating the need for explicit control ports and workflow-specific actors. Future extensions to support true semantic collection types will enable static analysis of workflows and will provide better support for cross-discipline and inter-organization sharing and reuse of workflow components.

The source code for the implementation described here is available in the latest release of KEPLER, which can be downloaded from the KEPLER project web site [12].

7. REFERENCES

[1] A. Ailamaki, Y. Ioannidis, and M. Livny. Scientific workflow management by database management. In *SSDBM*, 1998.

[2] C. Berkley, S. Bowers, M. Jones, B. Ludäscher, M. Schildhauer, and J. Tao. Incorporating semantics in scientific workflow authoring. In *SSDBM*, 2005.

[3] S. Bowers and B. Ludäscher. Actor-oriented design of scientific workflows. In *Proc. of the Intl. Conf. on Conceptual Modeling (ER)*, 2005.

[4] L. Bright and D. Maier. Deriving and managing data products in an environmental observation and forecasting system. In *Conf. on Innovative Data Systems Research (CIDR)*, 2005.

[5] 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.

[6] 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. *Concurrency and Computation: Practice and Experience, Special Issue on Scientific Workflows*, 2005.

[7] The office of science data-management challenge. Report from the DOE Office of Science Data-Management Workshops, March–May 2004.

[8] J. Felsenstein. *Inferring Phylogenies*. Sinauer Associates, Inc., 2004.

[9] L. Golab and M. T. Özsu. Issues in data stream management. *ACM SIGMOD Record*, 2003.

[10] P. Gordon. XML for molecular biology. http://www.visualgenomics.ca/gordonp/xml/.

[11] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(6), 2003.

[12] The Kepler Project. http://www.kepler-project.org.

[13] E. A. Lee and S. Neuendorffer. Actor-oriented models for codesign: Balancing re-use and performance. In *Formal Methods and Models for Systems*. Kluwer, 2004.

[14] E. A. Lee and T. M. Parks. Dataflow process networks. *Proc. of the IEEE*, 83(5):773–801, 1995.

[15] B. Ludäscher and I. Altintas. On providing declarative design and programming constructs for scientific workflows based on process networks. Technical report, SciDAC-SPA-TN-2003-01, 2003.

[16] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows*, 2005.

[17] D. Maddison, D. Swofford, and W. Maddison. NEXUS: An extensible file format for systematic information. *Systematic Biology*, 46(4), 1997.

[18] J. Morrison. *Flow-Based Programming*. Van Nostrand Reinhold, 1994.

[19] Natural Diversity Discovery Project. http://www.nddp.org.

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

[21] SciTegic. http://www.scitegic.com/.

[22] D. Weinstein, S. Parker, J. Simpson, K. Zimmerman, and G. Jones. *Visualization Handbook*, chapter Visualization in the SCIRun Problem Solving Environment. Elsevier, 2005.