Scientific Workflow Design 2.0: Demonstrating Streaming Data Collections in Kepler

Lei Dou*, Daniel Zinn*, Timothy McPhillips*, Sven Köhler*, Sean Riddle*, Shawn Bowers†, Bertram Ludäscher*

*UC Davis Genome Center, University of California, Davis, CA 95616, USA
{ldou,dzinn,tmcphillips,svkoehler,swriddle,ludaesch}@ucdavis.edu

†Department of Computer Science, Gonzaga University, Spokane, WA 99258, USA
bowers@gonzaga.edu

***Abstract*—Scientific workflow systems are used to integrate existing software components (actors) into larger analysis pipelines to perform *in silico* experiments. Current approaches for handling data in nested-collection structures, as required in many scientific domains, lead to many record-management actors (*shims*) that make the workflow structure overly complex, and as a consequence hard to construct, evolve and maintain. By constructing and executing workflows from bioinformatics and geosciences in the Kepler system, we will demonstrate how COMAD (Collection-Oriented Modeling and Design), an extension of conventional workflow design, addresses these shortcomings. In particular, COMAD provides a hierarchical data stream model (as in XML) and a novel declarative configuration language for actors that functions as a middleware layer between the workflow's data model (streaming nested collections) and the actor's data model (base data and lists thereof). Our approach allows actor developers to focus on the internal actor processing logic oblivious to the workflow structure. Actors can then be re-used in various workflows simply by adapting actor configurations. Due to streaming nested collections and declarative configurations, COMAD workflows can usually be realized as linear data processing pipelines, which often reflect the scientific data analysis intention better than conventional designs. This linear structure not only simplifies actor insertions and deletions (workflow evolution), but also decreases the overall complexity of the workflow, reducing future effort in maintenance.**

I. Introduction

Research in the area of scientific workflows received attention from the database community due to the structurally complex or large data involved, resulting in challenges for data management and efficient execution. Scientific workflow systems are used to build complex scientific analysis pipelines from existing software components (wrapped as *actors*) based on a dataflow-oriented paradigm. *Actors* have distinct *ports* over which they receive input data and emit output data. During workflow design, actors are placed on a canvas and output ports are linked to input ports to form the *workflow graph*, which specifies the dataflow during workflow execution.

In many scientific domains, data is structured hierarchically in (nested) collections. For example, meteorological data is often organized by station, county and state; bioinformatics data comprises nucleotides or amino-acid sequences, organized by genes, organisms, species, or larger groups. To maintain data associations, current workflow systems [3, 4, 6, 7, 9, 11] provide the capability to create user-defined record or list-data structures. However, workflows deploying these user-defined "collection types" necessarily contain many *shim* actors to assemble or disassemble these structures. For example, Lin et al. [8] report that the workflows in myExperiment include on average 30% shim actors. Using these shims in the workflow effectively couples the data-organization with the workflow graph. Resulting designs are thus not only hard to create and understand (many actors are not scientifically meaningful), but also hard to evolve (changes to the list or record structures often require major re-designs of the workflow structure). Providing programing abstraction to make scientific workflow design more effective is a major challenge, which COMAD takes a first step in addressing.

We will demonstrate[1] how Collection-Oriented Modeling and Design (COMAD) not only eliminates these shim actors, but also provides several other design properties (e.g., descriptive actor interfaces, decoupling of workflow graph and data structures). In COMAD, the hierarchical data is flattened into a token stream much like a SAX stream of XML data. During workflow execution, the data is streamed through the workflow from one actor to another. Similar to workers on conveyer belt assembly lines, actors in COMAD pick up certain data from the stream, process it and put the output back into the stream. All the data selection, validation, conversion, and insertion is performed automatically by the COMAD configuration layer. Configurations are expressed in a declarative actor scoping and binding language that uses XPath-like expressions.

In this work, we describe and demonstrate the key concepts of COMAD 2.0, which we recently released as a Kepler module[2]. These are (1) stream-based workflow execution, (2) nested, labeled collections of data with possible annotations, and the (3) scoping and binding configuration language. While our prior work has shown the overall effectiveness of nested collections for scientific workflow design [10, 13], and proposed early steps in adapting an XML-based configuration language [12], this work describes extensions to the data model, the configuration language, support for annotations, as well as the system architecture of our released implementation.

II. System Overview

A. Data Model: Streaming Nested Labeled Collections

The logical data model of COMAD consists of three entities: nested labeled collections, data items, and annotations.

[1]For a movie see http://youtube.com/watch?v=rYBbInsOonQ

[2]The open-source, BSD-licensed COMAD 2.0 module is available for domain and computer scientists via the Kepler Module Manager.

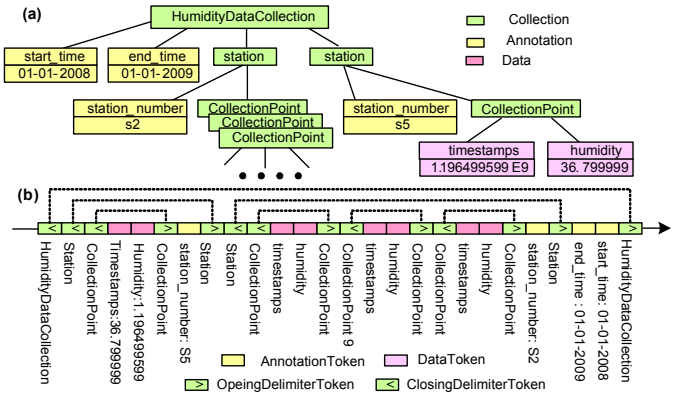

Fig. 1. Humidity data represented in the COMAD data model: Logical tree-structure (a), and corresponding physical token stream (b).

Collections are labeled containers for other entities including collections themselves leading to a tree-like data organization much like XML. Data items represent the domain-specific data, which can be of simple types like Integer or String as well as of user-defined Java-classes wrapping data such as Images, DNA sequences, or phylogenetic trees. Each data item has a type attribute, and an optional user-defined label. Furthermore, both collections and data items can have additional annotations associated with them. Annotations can be seen as special data items with a strong relationship to the annotated entity; their label (called *key*) is mandatory and unique amongst other annotations for the same data or collection. Furthermore, annotations of collections are inherited by enclosed data entities and nested collections, allowing these nested entities direct access to this meta-data. If nested collections or nested data items have an explicit annotation with the same key as an implicit, inherited annotation, then the explicit one overwrites the implicit one.

Physically, this data model is implemented as a *token stream*, similar to a SAX-like serialization of XML data. Each logical entity is mapped to specific tokens (DataToken, AnnotationToken, CollectionDelimiterToken). Provenance information about data, annotations, and collections is maintained via special annotation-tokens that are automatically inserted into the stream by the COMAD framework. Figure 1 shows an example of the logical structure and the physical stream-layout of humidity data collected from multiple weather stations in the COMET project [5].

The COMAD data model natively supports collection-oriented data, which is conventionally mapped to specific opaque types (built from arrays, lists or records). Having the data "disassembled" in the COMAD stream, obviates explicit assembly/disassembly shims. Instead, declarative actor configurations are used to denote which data is selected from the token stream. The actual stream processing to evaluate the "binding queries" is performed by the framework. Since data selection and packaging is performed by the framework, the graph structure of COMAD workflows is usually much simpler. Furthermore, since the data is staged for the actor by the framework, also actor development is simplified. By using annotations, additional data can be closely attached to collections and data items. Simple tags, for example, are commonly used in practice to select or group data accordingly for different analysis purposes. The COMAD configuration language allows to use annotations directly as input, or as conditions for filtering input data.

B. Actor Anatomy

The main components of a COMAD actor are its *black-box* and its *configurations*. The former implements the scientifically meaningful computations (e.g., data analysis), the latter declaratively describes the data handling via *black-box signature*, *scope*, and *data bindings*.

The **black-box signature** is defined by the actor developer. It declares the data inputs and outputs of the black-box, including data type and cardinality. Any system built-in type or user defined type can be used. We allow the usual options for cardinality: one, zero or one, one or more, and zero or more represented by no suffix, ?, +, and * respectively. For example, StringToken+ represents a non-empty list of type StringToken.

The **scope** parameter defines the subtrees (*scope matches*) inside the actor's input, over which the actor is *invoked*. Collections that are not inside scope matches are ignored by, i.e., passed through, the actor.

During each invocation, input/output **data bindings** select the concrete input data for the black-box from the scope match, and define where the black-box's output should be inserted into the data stream. While the scope points to a collection, data bindings point to data or annotation items. The data bindings must be consistent with the black-box signature, which is ensured by the COMAD framework during workflow design-time (if possible) and during runtime. The cardinality test, for example, guarantees that for each black-box *firing*, there will be data provided to all black-box inputs that disallow empty inputs. Furthermore, a type consistency test guarantees that bound data is compatible with the data input requested by the signature, and it guarantees that the black-box's output data is compatible with the data-type requested in the output data binding. Here, compatible means that either their types match, or there exists a lossless conversion, which will be performed automatically by the COMAD framework.

The distinction between signature and data bindings clearly separates the role of actor developer and workflow designer, and thus facilitates actor re-use. Actor development is reduced to implementing the core computation logic and defining an appropriate, workflow-independent signature. The workflow designer chooses actors from a library, and configures their data bindings according to the black-box signature and the concrete data stream of the workflow. Thus, actors can easily been adapted to workflows with different data organizations.

C. Actor Scoping and Binding Language

The actor scoping and binding language (SBL) is used to define the scope and data bindings of COMAD actors. It allows to specify data selection for black-box input, and data placement for black-box output. To enable more general stream transformations, SBL also supports the creation and

deletion of collections, data items and annotations. Via specific extensions to XPath, these operations can be specified based on the location within the collection/XML structure, existing annotations, and other conditions.

Based on XPath, SBL expressions consist of a path anchor, followed by axes and tests. Scope expressions are implicitly anchored at the root-collection, whereas data bindings are anchored to their respective scope matches. Like XPath, SBL uses '/' and '//' axes to denote child or descendent relationships within the collection structure. Label-tests within the path expressions are applied to collections (referring to the collection-label) and data items, where the test is by default applied to the type of the item. Our experience, gained from working with domain-scientists, shows that this coincides with the way scientific procedures are described: "Read the alignment data of the multiple sequences and output a Newick data item for the inferred phylogenetic tree." Annotations can be selected based on their key with a prepended @-symbol (like XML attributes in XPath). Qualifiers (e.g., referencing annotation and data values) can be used as conditional "side-axes" to select items more precisely. SBL is used to select input data for the black-box, as well as to specify where the output data is placed into the nested collection structure. To insert output items relative to where input was located or relative to other outputs destination, we introduced *port references* that can be used as an alternative anchor-point for the path expressions in output data bindings. To allow deletions of data items, collections and annotations from the data stream, a special *delete* decoration can be added to input data bindings, making it easy to define filter actors. Furthermore, *create* decorations can be used to create new collections per actor invocation, black-box firing, or as a peer collection of the scope matches, making it very easy to organize the output data in a flexible way. These features for data organization and collection-management have been proven to be very useful based on our work with domain-scientists.

D. Framework Architecture

Figure 2 shows the architecture of the COMAD framework. COMAD workflows can usually be built as linear workflows with data streaming through the actors like items on a physical assembly line. Each black-box is enclosed by a configuration layer that provides data management functionalities. The configuration layer comprises five modules. From bottom upwards they are the following:

Token Handler. Actor developers can choose to interact with the incoming data stream in an event-driven mode by registering custom token handlers. Also, the remaining four modules are implemented via token handlers.

Scope Matcher & Data Binder. This module determines scope matches, buffers actor input data according to its data bindings, and writes actor output data back into the token-stream. All tokens outside the scope directly bypass the actor.

Data Type Validator & Converter. This module tests type consistency between the bound data and the black-box signature. If consistent, data is converted to the expected type if needed and put into buffers.

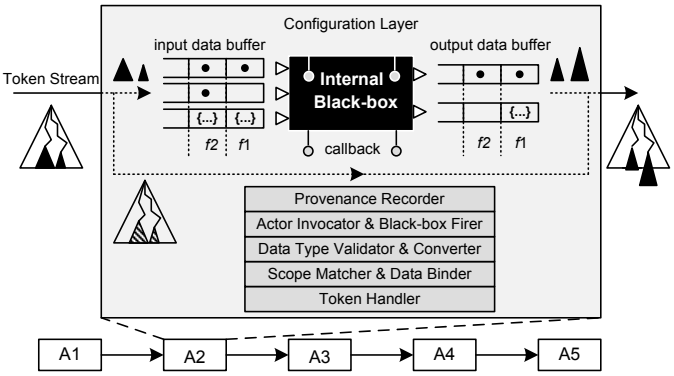

Fig. 2. Architecture of the COMAD Framework.

Actor Invocator & Black-box Firer. Each scope match constitutes one actor invocation. As soon as there is input data for all black-box inputs that disallow empty data, the black-box is executed (i.e., actor is *fired*) with this group of buffered data. If there are multiple groups of buffered data available during the actor invocation, the actor will be fired multiple times. The output of each firing (if existent) is inserted into the corresponding output buffers, which then, will be re-inserted into the stream by the scope matcher and data binder module.

Provenance Recorder. This module captures provenance information about creations and deletions of data items. By default, the dependencies of a created item are set to all the input data inside the current actor scope match that has been processed so far. These dependencies can be customized, for example to only depend on the input data of one firing, or a specific subset of these.

III. Demonstration Details

We will demonstrate actor development and workflow design based on several workflows originating from research performed in environmental science and bioinformatics.

A. COMET Workflow

This workflow analyzes meteorological data from weather stations in California in the context of the COMET project [5]. The workflow's main steps are: (1) collecting hourly humidity data from weather stations, (2) aggregating these data based on time windows and calculating statistical values for each window. Statistics include basic aggregates such as minimum, maximum, and average, as well as more complex ones like the "growing degree days" (Gdd). The final step in the workflow is (3) drawing a time-based trend graph. This demonstration will emphasize the following COMAD features:

Linear workflow structure. Like many COMAD workflows, this workflow exhibits a simple linear structure; see Fig. 3(a). Each actor picks up data from its input stream, processes it and outputs the results back to the data stream according to its scope and data bindings. The configurations for the GddCalculator actor are shown in Fig. 3(b): Its scope is `//AggregationResult/window`, where it binds the DoubleToken labeled with `min` and `max` to the corresponding

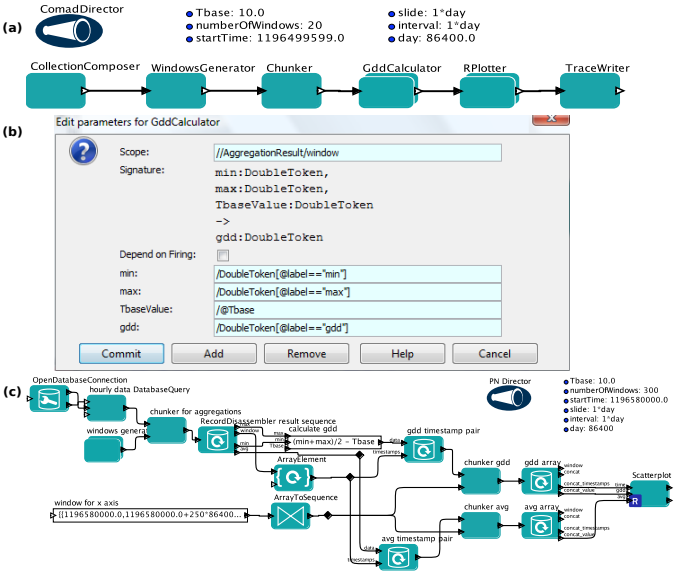

Fig. 3. COMET workflow built with the COMAD framework (a) and standard Kepler PN (c). (b) shows GDDCALCULATOR configuration.

input ports of the black-box component. The `Tbase` value is bound to an annotation, which can be defined at a higher collection-level (e.g., as a global configuration on the overall root collection). The annotation is accessible here due to the inheritance semantics of annotations. The computed growing degree days value is then inserted as a new DoubleToken with label `gdd` under the `window` collection. Fig. 3(c) shows a functional equivalent workflow built using a conventional process network (PN) model. Note that the linear structure of the COMAD workflow more clearly reflects the data analysis than the PN workflow, which contains record/array management shims besides the scientifically meaningful actors.

Streaming mode. To demonstrate the streaming execution model of COMAD workflows, we will add a SEQUENCEPLOTTER actor after the original plotter, RPLOTTER. SEQUENCEPLOTTER implements a plotter that dynamically updates its display as new data is streamed into its input port. Our demonstration will show that SEQUENCEPLOTTER begins plotting long before RPLOTTER although it is placed behind RPLOTTER in the workflow. This behavior is because the RPLOTTER actor does not plot until all data is received; but, since the data stream is passed on, the SEQUENCEPLOTTER can display the data as it arrives incrementally.

High reusability and adaptability of COMAD actors. We will demonstrate the adaptability of COMAD actors and workflows by demonstrating necessary workflow changes to cope with input data changes (e.g., renaming collections, and adding more hierarchies). Originally, data from one station is used, then from multiple stations within one county, and finally from stations of multiple counties. The workflow adapts to these changes without any necessary changes to the configurations. Only if we rename input collections do we need to change the scope or binding parameters of affected actors.

Ease of actor development. We will walk through the source-code of the WINDOWSGENERATOR actor. Here, only the window generating logic is implemented. No additional code has to be written to interact with the collection-oriented stream, neither for fetching inputs nor for placing outputs.

Provenance recording. Provenance is recorded by the framework while the data stream passes through the workflow actors. We will use the ProvenanceBrowser [1] to show the recorded provenance and demonstrate the various dependency policies, e.g., per-scope, per-firing, or black-box-defined.

B. BioMoby Workflows

A COMAD actor is configurable via its scope and input bindings; even the black-box component itself and its signature can be configured. Therefore, groups of similar services can be mapped to a single COMAD actor instead of developing a large number of actors with similar behavior. We will demonstrate this via an actor we created to wrap BioMoby [2] services. After instantiation with the service name as a parameter, black-box signatures (including data-types and cardinalities) are automatically configured based on the published BioMoby service description. We will demonstrate two workflows composed of BioMoby services: A BLAST workflow (to find DNA sequences that are similar to an input sequence), as well as a phylogenetics workflow which infers phylogenetic trees from a group of input DNA or protein sequences.

Acknowledgements. This work was supported in part by NSF awards DBI-0960535, OCI-0722079, AGS-0619139, and DOE DE-FC02-07ER25811.

REFERENCES

[1] M. K. Anand, S. Bowers, and B. Ludäscher. Provenance browser: Displaying and querying scientific workflow provenance graphs. In *ICDE*, pages 1201–1204, 2010.
[2] Biomoby. http://www.biomoby.org.
[3] S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. Vo. VisTrails: visualization meets data management. In *SIGMOD'06*, pages 745–747. ACM, 2006.
[4] 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*, 18(10):1021–1037, 2006.
[5] Comet project. http://comet.ucdavis.edu.
[6] E. Deelman, G. Singh, M. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.
[7] T. Fahringer, R. Prodan, R. Duan, F. Nerieri, S. Podlipnig, J. Qin, M. Siddiqui, H. Truong, A. Villazon, and M. Wieczorek. ASKALON: A Grid Application Development and Computing Environment. *International Workshop on Grid Computing*, pages 122–131, 2005.
[8] C. Lin, S. Lu, X. Fei, D. Pai, and J. Hua. A task abstraction and mapping approach to the shimming problem in scientific workflows. In *IEEE Intl. Conf. on Services Computing*, Bangalore, India, 2009.
[9] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, 2006.
[10] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541–551, 2009.
[11] 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.
[12] D. Zinn, S. Bowers, and B. Ludäscher. Xml-based computation for scientific workflows. In *ICDE'10*, pages 812–815, 2010.
[13] D. Zinn, S. Bowers, T. M. McPhillips, and B. Ludäscher. Scientific workflow design with data assembly lines. In *WORKS'09*, 2009.