Scientific Workflow Design for Mere Mortals

Timothy McPhillips, Shawn Bowers, Daniel Zinn, Bertram Ludäscher

University of California at Davis
One Shields Avenue
Davis, CA 95616, U.S.A.

Abstract

Recent years have seen a dramatic increase in research and development of scientific workflow systems. These systems promise to make scientists more productive by automating data-driven and compute-intensive analyses. Despite many early achievements, the long-term success of scientific workflow technology critically depends on making these systems useable by "mere mortals", i.e., scientists who have a very good idea of the analysis methods they wish to assemble, but who are neither software developers nor scripting-language experts. With these users in mind, we identify a set of desiderata for scientific workflow systems crucial for enabling scientists to model and design the workflows they wish to automate themselves. As a first step towards meeting these requirements, we also show how the *collection-oriented modeling and design* (COMAD) approach for scientific workflows, implemented within the Kepler system, can help provide these critical, design-oriented capabilities to scientists.

1. Introduction

Scientific workflow technology has emerged over the last few years as a challenger to long-established approaches to automating computational tasks. Due to the wide range of analyses performed by scientists, however, and the diverse requirements associated with their automation, scientific workflow systems are forced to address an enormous variety of complex issues. This situation has led to specialized approaches and systems that focus on particular aspects of workflow automation, such as workflow deployment within high-performance computing and Grid environments [TBF+06,DSS+05,PA06,FPD+05], fault-tolerance and recovery [TTK+07,ABJF06,HK03], workflow composition languages [GRD+07,SdSGS06,BL05], workflow specification management [DRGS07,WGG+07], and workflow and data provenance [GLM04,BCS+05,ZWF06,SPG05]. A far smaller number of systems have been developed explicitly to provide generic and comprehensive support for the various challenges associated with scientific workflow automation (e.g., [LAB+06,MSTW04,OGA+06]).

The intended users of many of these systems (particularly the latter, more comprehensive ones) are scientists who are expected to interact directly with the systems to design, configure, and execute scientific workflows. Consequently, the long-term success of such scientific workflow systems critically depends on making these systems not only useful to scientists, but also directly useable by them. As such, these systems must provide scientists with explicit and effective support for workflow modeling and design. Regardless of how a workflow is ultimately deployed—within a local desktop computer, web server, or distributed computing environment—scientists must have models and tools for designing scientific workflows that correctly and efficiently capture their desired analyses. In this paper we identify important requirements for scientific workflow systems and present COMAD, a workflow modeling and design framework that aims to address these needs.

Scripting Languages for Tool Integration. Many scientists today make extensive use of batch files, shell scripts, and programs written in general-purpose scripting languages (e.g., Perl, Python) to automate their *tool-integration* tasks. Such programs typically combine and chain together sequences of heterogeneous applications for processing, manipulating, managing, and visualizing data. These generic scripting languages are often distinguished from more specialized languages, computing platforms, and data analysis environments (e.g., R, SAS, Matlab), which target scientific users with more sophisticated needs (e.g. data analysts, algorithm developers, and researchers developing new computational methods for particular domains). Many of these more specialized scientific computing platforms now provide support for interacting with and automating external applications, and domain-specific libraries are increasingly being developed for use via scripting languages (e.g., BioPerl [1]). Thus, for scientific workflow systems to become broadly adopted as a technology for assembling and automating analyses, these systems must provide scientists concrete and demonstrable advantages, both over general-purpose scripting languages and more focused scientific computing environments currently occupying the tool-integration niche.

Scientific Workflow Systems. Existing scientific workflow systems generally share a number of common goals and characteristics [GDE+07] that differentiate them from tool-integration approaches based on scripting languages and other platforms with tool-automation features. One of the most significant differences is that whereas scripting approaches are largely based on imperative languages, scientific workflow systems are typically based on *dataflow languages* [JHM04,GDE+07] in which workflows are represented as directed graphs, with nodes denoting computational steps (or *actors*), and connections representing data dependencies (and data flow) between steps. Many

Email addresses: tmcphillips@ucdavis.edu (Timothy McPhillips), sbowers@ucdavis.edu (Shawn Bowers), dzinn@ucdavis.edu (Daniel Zinn), ludaesch@ucdavis.edu (Bertram Ludäscher).

[1] http://www.bioperl.org

Preprint of Journal Article http://dx.doi.org/10.1016/j.future.2008.06.013

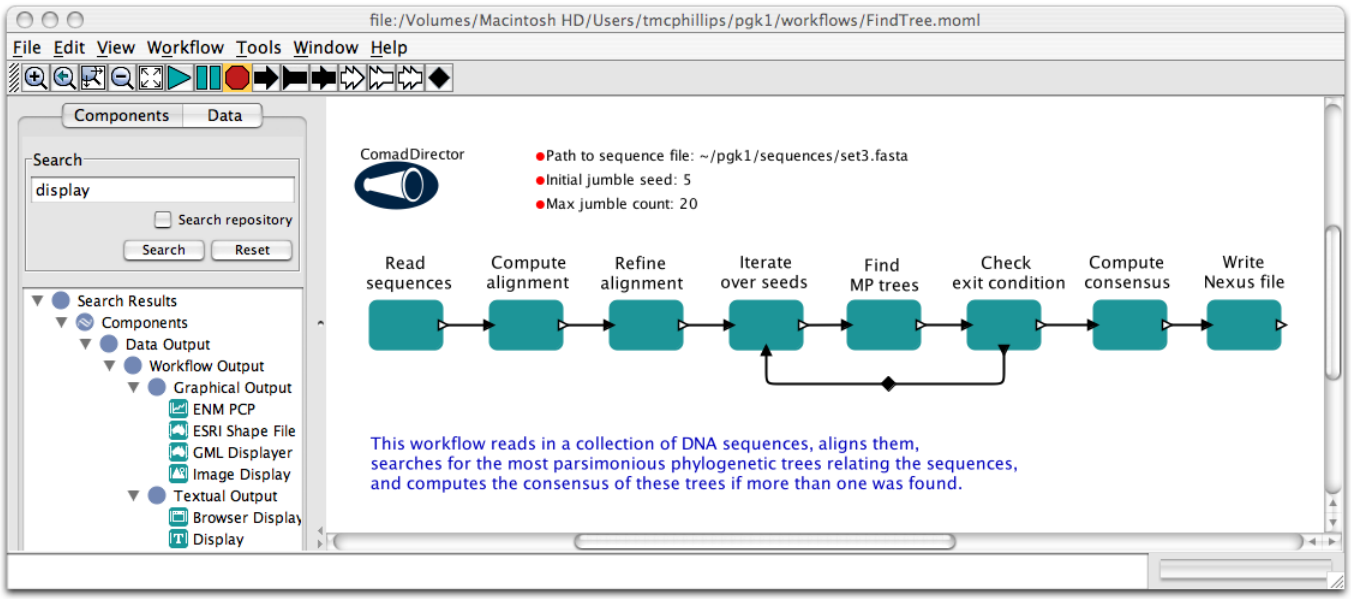

Fig. 1. A phylogenetics workflow implemented in the Kepler system. Kepler workflows are built from *actors* (boxes) that perform computational tasks. Users can select actors from component libraries (panel on the left) and connect them on the canvas to form a workflow graph (center/right). Connections specify dataflow between actors. Configuration parameters can also be provided (top center), *e.g.*, the location of input data and the initial *jumble seed* values are given. A *director* (top left corner on the canvas) is a special component, specifying a model of computation and controlling its execution.

systems (e.g., [BCS+05,LAB+06,MSTW04,OGA+06]) allow workflows to be created and edited using graphical interfaces (see Fig. 1 for an example in Kepler). The dataflow paradigm is well-suited for supporting modular workflow design and facilitating reuse of components [JHM04,LP95,LAB+06,BL05]. Many workflow systems (e.g., [OGA+06,LAB+06]) further allow workflows to be used as actors in other workflows, thus providing workflow authors an abstraction mechanism for hiding implementation details and facilitating even more reuse.

One advantage of workflow systems that derives from this dataflow-orientation is the ease with which data produced by one actor can be routed to multiple downstream actors. While the flow of data to multiple receivers is often difficult to describe clearly in plain text, the dataflow approach makes explicit this detailed routing of data. For instance, in Fig. 1 it is clear that data can flow directly from Refine alignment only to Iterate over seeds. The result is that scientific workflows can be more declarative about the interactions between actors than scripts, where the flow of data between components is typically hidden within (often complex) code. The downside of this approach is that if taken too far, specifications of complex scientific workflows can become a confusing tangle of actors and wires unless the workflow specification language provides additional, more sophisticated means for declaring how data is to be routed (as COMAD does—see below as well as [MBL06] and [BLNC06]).

Other notable advantages of scientific workflow systems over traditional approaches are their potential for transparently optimizing workflow performance and automatically recording data and process provenance. Unlike most scripting language implementations, scientific workflow systems often provide capabilities for executing workflow tasks concurrently where data dependencies between tasks allow, either in an "assembly-line" fashion with actors connected in a linear pipeline performing their tasks simultaneously, or in parallel with multiple such pipelines operating at the same time (e.g., over multiple input data sets or via explicit branches in the workflow specification) [YB05,PA06,MBL06]. Many scientific workflow systems also can record, store, and query data and process dependencies that result during one or more workflow runs, enabling scientists to later investigate the data and processes used to derive results and to examine intermediate data products [SPG05,ML08].

While these and other advantages of systems designed specifically to automate scientific workflows help to position these technologies as viable alternatives to traditional approaches based on scripting languages and the like, much is yet required to achieve the vision of putting workflow automation fully into the hands of "mere mortals" [GDE+07]. Much remains to be done to realize the vision of scientists untrained in programming and relatively ignorant of the details of information technology rapidly composing, deploying, executing, monitoring, and reviewing the results of scientific workflows without assistance from information-technology experts.

Contributions and Paper Outline. In this paper we describe key aspects of scientific workflow systems that can help enable broader-scale adoption of workflow technology by scientists, and demonstrate how these properties can be realized by a novel and generic workflow modeling paradigm that extends existing dataflow computation models. In Section 2, we present what we see as important desiderata for scientific workflow systems from a workflow modeling and design perspective. In Section 3, we describe our main contribution, the *collection oriented modeling and design* (COMAD) framework, for deliver-

ing on the expectations described in Section 2. Our framework is especially suited for cases where data is nested in structure and computational steps can be pipelined (which is often true, e.g., in bioinformatics). The COMAD framework provides an assembly-line style computation approach that closely follows the spirit of flow-based programming [Mor94]. The COMAD framework has been implemented as part of the Kepler system [LAB+06] and has been successfully used to implement a range of scientific workflows. Finally, we discuss related work in Section 4 and conclusions in Section 5.

The goal of this paper is not to show that our approach is the best way to implement all scientific workflows, but rather to demonstrate that the ambitious-sounding requirements commonly attached to scientific workflows and spelled out explicitly in Section 2 can largely be satisfied by an approach applicable to a range of scientific domains. We hope in this way to inspire others to further identify and tackle head-on the challenges to wide-scale adoption of scientific workflow systems by the scientific community.

2. Desiderata for Scientific Workflow Systems

The following desirable characteristics of scientific workflow systems are targeted at a specific set of users, namely, *researchers in the natural sciences developing their own scientific workflows to automate and share their analyses*. For these users to benefit from scientific workflows, we believe workflow systems should distinguish themselves from scripting languages and other general purpose tools in three principal ways: (1) they should help scientists design and implement workflows; (2) they should provide first-class support for modeling and managing scientific data, not just analytical processes; and (3) they should take responsibility for optimizing performance. Within these three categories we argue for eight specific desiderata for scientific workflow systems.

The desiderata presented below are based on our own experiences working with scientists through various projects aimed at implementing scientific workflows and developing supporting workflow technology. These desiderata largely arrise from issues concerning workflow modeling and design, and in the following section we describe how these requirements can be satisfied using the COMAD approach (see Fig. 2). While existing scientific workflow systems support some or all of these desiderata in a variety of ways (see [YB05] and Section 4), we focus below on the capabilities and limitations of the Kepler scientific workflow system, which provides the framework and context for most of our work.

2.1. *Assist in the Design and Implementation of Workflows*

Scientific workflow systems such as Kepler expect the user to compose workflows incrementally, selecting modules from a library of installed components and wiring the components together. Kepler currently helps the user during the workflow design process in a number of ways. For example, Kepler enables powerful keyword searches over actor metadata and ontology annotations to quickly find relevant actors in local or distributed libraries [BL05]. Similarly, subworkflows can be encapsulated as composite actors within Kepler workflows, and output data types of one actor can be checked against the expected input types of another actor. However, workflow systems are ideally placed to do much more to make it easy to design workflows.

Well-Formedness: *Workflow systems should make it easy to design well-formed and valid workflows.* (WFV)

Workflow systems should be able to detect when workflows do not make sense overall, or when parts of the workflow will not contribute to the result of a run. Similarly, workflow systems should enable users to declare the types of the expected inputs and outputs of a workflow, and ensure well-formedness by verifying that all workflow actors and input data items will indeed contribute to the production of the expected workflow products.

The reason for this is that scientific workflows are much more like recipes used in the kitchen, or protocols carried out in a lab, than is the average computer program. Workflows are meant to produce well-defined results from well-defined inputs, using well-defined procedures. Few scientists would commit to carrying out an experimental protocol that does not make clear what the overall process will entail, what products (and how much of each) the protocol is meant to yield, and how precisely that product will be obtained (see *clarity* below). Scientists would be justified in being equally dubious about a "scientific" workflow that is not as clear and predictable as the protocols they carry out in the lab. They should be particularly worried when the workflows they design are so obscure as to be not predictable in this way (see *predictability* below).

Clarity: *Workflow systems should make it easy to create self-explanatory workflows.* (CLR)

Any scientist composing a new workflow will have a fairly good idea of what the workflow should do when it executes. Ideally the system would confirm or contradict this expectation and thus provide immediate feedback to the scientist. In current systems, however, expectations about what will happen during a run often can only be checked by running a workflow on real data and checking if the results look reasonable. Because an actual run may be impractical to execute while developing a workflow, either because the run would take too long or because the required computational resources cannot be spared, understanding the behavior of a workflow without running it would facilitate workflow design immensely.

One solution to this problem would be to make the language for specifying workflows so clear and declarative that a scientist could tell at a glance what a workflow will do when executed. This in turn requires that systems provide scientists with workflow abstractions relevant to their domain. Instead of enmeshing users in low-level details that obscure the scientific meaning of the workflow, systems should provide abstractions that hide these technical details, especially those details that have more to do with information technology than the particular scientific domain.

Predictability: *Workflow systems should make it easy to understand what a workflow will do before running it.* (PRE)

Unfortunately, the complexities of data management and the need for iteration and conditional control-flow often make it difficult to foresee the complete behavior of a workflow even when the workflow is defined in terms familiar to the user. In these cases where the function of a workflow cannot be read directly from the workflow graph, systems need to be able to predict, in some way that is meaningful to a scientist, what will happen when a workflow is run.

Workflow systems should also make it easy for collaborators to understand the purpose and expected products of a workflow. Many scientific projects involve multiple collaborators that rely on each other's data products. Understanding data in such projects often requires understanding the analyses involved in producing the data. Thus, scientific workflow designs should also make it possible to quickly and easily understand the steps involved in an analysis by someone *other* than the creator of the workflow.

Recordability: *Workflow systems should make it easy to see what a workflow did do when it ran.* (REC)

Understanding workflow behavior after it occurs is often more important to scientists than predicting workflow behavior in advance. There is no point in carrying out a "scientific" analysis if one cannot later determine how results were obtained. Unfortunately, for various reasons, recording what happened within a workflow run is not as easy as it sounds. For instance, due to parallel and concurrent optimizations, the "raw" record of workflow execution will likely be as difficult to interpret as, e.g., a single log-file written to by multiple Java threads. There also are numerous types of events that can be recorded by a system, ranging from where and when a workflow was run, to the amount of time taken and memory used by each invocation (execution) of an actor, all the way down to the low-level details of what hardware and software configurations were used during workflow execution. The latter details are useful primarily to engineers deploying workflow systems and troubleshooting performance problems. For scientists what is most needed are approaches for accurately recording actor invocation events and associating these with the data objects consumed and produced during each such that the scientific aspects of workflow runs can be reviewed later.

Reportability: *Workflow systems should make it easy to see if a workflow result makes sense scientifically.* (REP)

Scientisits not only need to understand what data processing events occurred in a workflow run, but also how the products of the workflow were derived, from a scientific point of view, from workflow inputs. It is critical that this kind of data "lineage" information not distract the scientist with technical details having to do with how the workflow was executed. For example, it is not helpful to see that a particular sub-analysis was carried out at 11:39 PM on a particular node in the departmental Linux cluster when one is curious what DNA sequences were used to infer a particular phylogenetic tree. Instead, one would hope that a scientist reviewing the results of a run of the workflow in Fig. 1, e.g., could immediately see that the final phylogenetic tree was computed directly from five other trees via an invocation of the Compute consensus actor; that each of these trees were in turn computed from a sequence alignment via invocations of the Find MP trees actor; and so on. Such depictions of data dependencies often are referred to as *data lineage graphs* [ML08] and can be more effective as means for communicating the scientific justification for a computed result than the workflow specification itself.

Reusability: *Workflow systems should make it easy to design new workflows from existing workflows.* (REU)

Workflow development often means starting from an existing workflow. Workflow systems should minimize the work needed to reuse and repurpose existing workflows as well as help prevent and reveal the errors that can arise when doing so. Note that with many programming tools it is often easier and less error-prone to start afresh, rather than to refactor existing code. We can do better than this if we provide scientists with the design assistance features described here.

In a similar way, it is important to make it easy for scientists to develop workflows in a manner compatible with and supportive of their actual research processes. In particular, scientific projects are often exploratory in nature and the specific analyses of a project hard to predict *a priori*. Workflows must be easy to modify (e.g., by allowing new parameterizations, new input data, and new methods to be incorporated), chain together and compose, and track (i.e., to see in what context they were used, with what data, etc). Furthermore, support should be provided for designing workflows spanning a broad range of complexity, from those that are small and comprising only a few atomic tasks, to large workflows with many tasks and subworkflows.

2.2. *Provide First-Class Support for Modeling Data*

Scientists tend to have a data-centric view of their analyses. While the computational steps in an analysis certainly are important to scientists, they are not nearly as important as the data scientists gather, analyze, and create via their analyses. In contrast, current scientific workflow systems, including Kepler, tend to emphasize the *process* of carrying out an analysis. Although workflow systems enable scientists to perform powerful operations on data, they often provide only crude and low-level constructs for explicitly modeling data.

One consequence of this emphasis on process specifications (frequently at the expense of data modeling constructs) is that many useful opportunities for abstraction are missed. For example, if workflow systems require users to model their DNA sequences, alignments, and phylogenetic trees as strings, arrays, and other basic data types, then many opportunities for helping users design, understand, and repurpose workflows are lost.

Scientific Data Modeling: *Workflow systems should provide data modeling and management schemes that let users represent their data in terms meaningful to them.* (SDM)

One solution is to enable actor developers to declare entirely new data types specific to their domains, thus making it easier to represent complex data types, to hide their internal structure, and to provide intuitive abstractions of these types to the scientist composing workflows.

Another approach often used in workflow systems is to model data according to the corresponding file formats used for representation and storage (thus, file formats serve as data types). An actor in this case might take as input a reference to a file containing DNA sequences in FASTA format [2], align these sequences, and then output the alignment in the ClustalW format [THG94]. The biggest problem with this approach is that many file formats do not map cleanly onto individual data entities or simple collections of such entities. For example, a single file in Nexus format [MSM97] can contain phylogenetic character descriptions, data matrices, phylogenetic trees, and a wide variety of specialized information. It is very difficult to guess the function of a workflow module that takes one Nexus file as input and produces another Nexus file as output, or to verify automatically the meaningfulness of a workflow employing such an actor. It would be far better if workflow systems enabled modules to operate on scientifically meaningful types (as described above), and transparently provided application-specific files to the programs and services they wrap. Doing so would both help preserve the clarity of workflows and greatly enhance the interoperability of modules wrapping applications that employ different data formats.

Many application-specific file formats in science are meant primarily to maintain associations across collections of related data. A FASTA file can define a set of biological sequences. A Nexus file can store and declare the relationships between phylogenetic data matrices and trees inferred from them. Workflow systems also must provide ways of declaring and maintaining such associations without requiring module authors to design new, complex data types each time they run into a new combination of data items that must be operated on or produced together during a workflow. For example, a domain-specific data type representing a DNA sequence is useful to have, but it would be onerous to require that there be another custom data type representing a *set* of DNA sequences. Thus, workflow systems should provide generic constructs for managing collections of data.

Workflow systems that lack explicit constructs for managing collections of data often lead to "messy" workflows containing either many connections between actors to communicate the size of lists produced by one actor to actors consuming these lists; or many data assembly and disassembly actors; or both. The consequence of such *ad hoc* approaches for maintaining data associations during workflow runs is that the modeling of workflows and the modeling of data become inextricably intertwined. This leads to situations in which the structure of the *data* processed by a workflow is itself encoded implicitly in the workflow specification—and nowhere else.

Workflow systems should clearly separate the modeling and design of data flowing through workflows from the modeling and design of the workflow itself. Ideally, the workflow definition would specify the scientifically meaningful steps one wants to carry out; the data model would specify how the data is structured and organized, as well as how different parts of data structures are related to each other; and the workflow system would figure out how to carry out the workflow on data structured according to the given data model. While this may sound difficult to achieve, the closer we can get to achieving this separation the better it will be for scientists employing workflow systems.

[2] `http://www.ncbi.nlm.nih.gov/blast/fasta.shtml`

2.3. *Take Responsibility for Optimizing Performance*

Much of the impetus for developing scientific workflow systems derives from the need to carry out expensive computational tasks efficiently using available and often distributed resources. Workflow systems are used to distribute jobs, move data, manage multiple processes, and recover from failures. Existing workflow systems provide support for carrying out some or all of these tasks either explicitly, as part of workflow deployment, or implicitly, by including these tasks within the workflow itself. The latter approach is often used today in Kepler, resulting in specifications that are cluttered with job-distribution constructs that hide the scientific intent of the workflow. Workflows that confuse systems management with scientific computation are difficult to design in the first place and extremely difficult to re-deploy on a different set of resources. Even worse, requiring users to describe such technical details in their workflows excludes many scientists who have neither the experience nor interest in playing the role of a distributed operating system.

Automatic Optimization: *Workflow systems should take responsibility for optimizing workflow performance.* (OPT)

Even when workflows are to be carried out on the scientist's desktop computer, performance optimizations frequently are possible. However, systems should not require scientists to understand and avoid concurrency pitfalls—deadlock, data corruption due to concurrent access, race conditions, etc.—to take full advantage of such opportunities. Rather, workflow systems should safely exploit as many concurrent computing opportunities as possible, without requiring users to understand them. Ideally, workflow specifications would be abstract and employ metaphors appropriate to the domain rather than including explicit descriptions of data routings, flow control, and pipeline and task parallelism.

3. Addressing the Desiderata with COMAD

In this section, we describe how the *collection-oriented modeling and design* (COMAD) framework promises to make it easier for scientists to design workflows, to clearly show how workflow products were derived, to automatically optimize the performance of workflow execution, and otherwise make scientific workflow automation both accessible and practical for scientists. We also detail specific technical features of COMAD

Workflow System Desiderata	*Workflow & System Features Enabled by COMAD*
Clarity (CLR). Make it easy to create self-explanatory workflows.	Explicit actor read- and write-scope configurations require fewer 'shims' or 'adapters' that often clutter conventional workflow models. Workflows consist of actors that correspond to scientifically meaningful tasks.
Well-Formedness (WFV). Make it easier to design well-formed and valid workflows.	Workflow system can statically determine *actor relevance* via COMAD *type propagation*.
Predictability (PRE). Make it easy to see what a workflow will do without running it.	Workflow system can statically infer *output schema*, *actor dependencies*, and *schema lineage graph* showing data transformations before runtime; *minimal input schema* can also be inferred.
Recordability (REC). Make it easy to see what a workflow actually did do when it ran.	Workflow system can record "detailed" *provenance* of all relevant events in COMAD, e.g., actor invocations and associated inputs and outputs.
Reportability (REP). Make it easy to see if a workflow result makes sense scientifically.	Workflow system can provide *provenance* of data products by deriving an intuitive data-dependency graph from the detailed provenance records.
Reusability (REU). Make it easy to design new workflows from existing ones.	Read- and write-scope *configuration* and *part-of subtyping* make workflows more *change-resilient* with respect to input schema changes and actor insertions and deletions.
Scientific Data Modeling (SDM). Provide first-class support for modeling scientific data.	*Nested collections* (XML-like tree structures) are the basic data-modeling abstraction in COMAD, with explicit support for user-defined data types.
Automatic Optimization (OPT). Systems should take responsibility for optimizing performance.	Workflow system can provide *task-* and *pipeline-parallelism* (streaming data, concurrent actors). Additional support for *data shipping optimization* by exploiting schema (type) information.

Fig. 2. Desiderata for scientific workflow systems (from the perspective of a scientist wishing to automate and share their scientific analyses) and the COMAD features addressing these desiderata.

to show how it realizes the desiderata explicated above. Fig. 2 summarizes the COMAD features described here and how they relate to the desiderata of Section 2.

3.1. *An Introduction to* COMAD

As mentioned in Section 1, the majority of scientific workflow systems represent workflows using dataflow languages. The specific dataflow semantics used, however, varies from system to system [YB05]. Not only do the meaning of nodes, and of connections between nodes, differ, but the assumptions about how an overall workflow is to be executed given a specification can vary dramatically. Kepler makes explicit this distinction between the workflow graph, on the one hand, and the model of computation used to interpret and enact the workflow on the other, by requiring workflow authors to specify a *director* for each workflow (see Fig. 1). It is the director that specifies whether the workflow is to be interpreted and executed according to a process network (PN), synchronous dataflow (SDF), or other model of computation [LSV98].

Most Kepler actors in PN or SDF workflows are *data transformers*. Such actors consume data tokens and produce *new* data tokens on each invocation; these actors operate like functions in traditional programming languages. Other actors in a PN workflow can operate as filters, distributors, multiplexors, or otherwise control the flow of tokens between other actors; however, the bulk of the computing is performed by data transformers.

Virtual assembly-lines. In COMAD, the meanings of actors and connections between actors are different from those in PN or SDF. Instead of assuming that actors consume one set of tokens and produce another set on each invocation, COMAD is based on an assembly-line metaphor: COMAD actors (*coactors* or simply *actors* below) can be thought of as workers on a virtual assembly-line, each contributing to the construction of the workflow product(s). In a physical assembly line, workers perform specialized tasks on products that pass by on a conveyor belt. Workers only "pick" relevant products, objects, or parts thereof, and let all irrelevant parts pass by. Coactors work analogously, recognizing and operating on data relevant to them, adding new data products to the data stream, and allowing irrelevant data to pass through undisturbed (see Fig. 3). Thus, unlike actors in PN and SDF workflows, actors are *data preserving* in COMAD. Data *flows through* serially connected coactors rather than being consumed and produced at each stage.

Streaming nested data collections. A number of advantages can be gained by adopting an assembly-line approach to scientific workflows. Possibly the biggest advantage is that one can put information into the data stream that could be represented only with great difficulty in plain PN or SDF workflows. For example, COMAD embeds special tokens within the data stream to delimit collections of related data tokens. Because these *delimiter tokens* are paired, much like the opening and closing tags of XML elements (as shown in Fig. 3), collections can be nested to arbitrary depths, and this generic collection-management scheme allows actors to operate on collections of elements as easily as on single data tokens. Combined with an extensible type system, this feature satisfies many of the data modeling needs described in Section 2. Similarly, *annotation tokens* can be used to represent metadata for collections or individual data tokens, or for storing within the data stream the provenance of items inserted by coactors (see Fig. 3). The result is that coactors effectively operate not on isolated sets of input tokens, but on well-defined, information-rich collections

of data organized in a manner similar to the tree-like structure of XML documents.

3.2. *A Closer Look at COMAD*

Here we take a technical look at some features of COMAD, illustrating how this approach makes significant progress towards satisfying the desiderata described above.

Actor configurations and scopes. Assume we want to place an actor A in a workflow where the step before A produces instances of type τ and the subsequent step requires data of type τ':

$$\xrightarrow{\tau} \boxed{A: \alpha \to \omega} \xrightarrow{\tau'}$$

In the notation above, $A : \alpha \to \omega$ is the signature of actor A such that A consumes instances of type α and produces instances of type ω. Conventional approaches require that the type τ be a subtype of α and that ω be a subtype of the type τ', denoted $\tau \prec \alpha$ and $\omega \prec \tau'$. Often these type constraints will not be satisfied when designing a workflow, and adapters or shims must be added to the workflow as explained below.

In COMAD, we would instead model A as a *coactor*:

$$\xrightarrow{\tau} \boxed{\Delta_A : \tau_\alpha \to \tau_\omega} \xrightarrow{\tau'}$$

where an actor *configuration* $\Delta_A : \tau_\alpha \to \tau_\omega$ describes the *scope of work* of A. More specificially, Δ_A is used (i) to identify the *read-scope* τ_α of A, i.e., the type fragments relevant for A, and (ii) to indicate the *write-scope* τ_ω of A, i.e., the type of the new output fragments (if any). In addition, the configuration Δ_A needs to prescribe (iii) whether the type fragments matching τ_α are consumed (removed from the input stream) or kept, and (iv) where the τ_ω results are located within τ'.

These ideas are depicted in Fig. 4, where the relevant fragments matching τ_α are shown as black subtrees. These are consumed by actor A and replaced by A's outputs (of type τ_ω).

Clarity (CLR) and Reusability (REU). In Fig. 5 we illustrate a number of issues associated with designing declarative (clear) and reusable workflows, two of the desiderata discussed in Section 2. Conventional workflows tend to clutter a scientist's conceptual design (Fig. 5a) with lower-level glue actors, thus making it hard to *comprehend* and *predict* a workflow's behavior (Fig. 5b–d). Similarly, workflow *reuse* is made more difficult: when viewed in the context of workflow evolution, conventional workflows tend to be more "brittle", i.e., break easily as new actors are added or exisiting ones are replaced. As mentioned above, a conventional actor A can be seen as a data transformer, i.e., a function $A : \alpha \to \omega$. In Fig. 5a, each actor maps an input type α_i to an output type ω_i. The connection from A_i to A_{i+1} must satisfy the *subtyping constraint* $\omega_i \prec \alpha_{i+1}$. This rigid typing approach leads to the introduction of *adapters* [BL05], *shims* [HSL+04,RLSR+06], and to complex data- and control-flow constructs to send the exact data fragments to the correct actor ports, while ensuring type safety.

For example, suppose we want to add to the end of the conceptual pipeline in Fig. 5a, the new actor $A_4 : \alpha_4 \to \omega_4$.

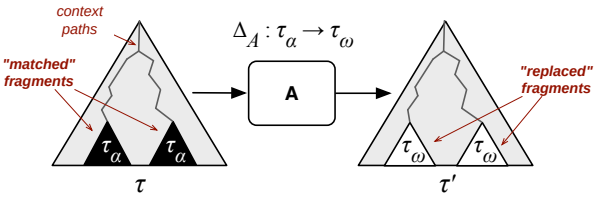

Fig. 4. The scope of actor (stream processor) A is given by a *configuration* Δ_A with read-scope τ_α (selecting relevant input fragments for A) and write-scope τ_ω (for A's outputs). Outputs replace inputs "in context".

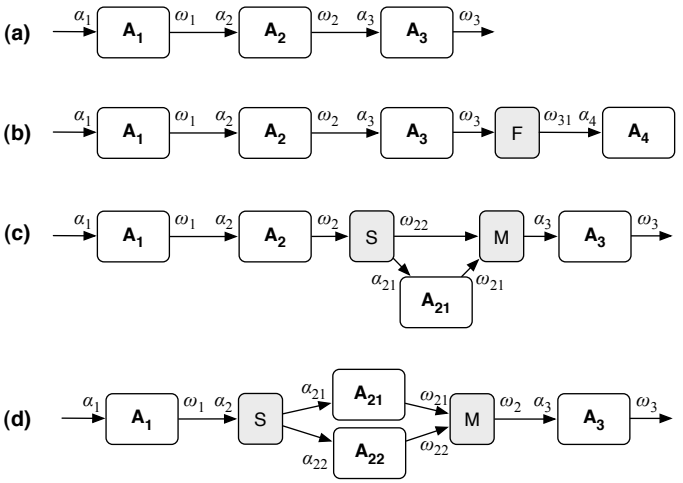

Fig. 5. Conventional workflows are rarely the simple analysis pipelines that scientists desire (a), but often require "glue" steps (*adapters*, *shims*), cluttering and obfuscating the scientists' conceptual design, leading to workflows that are difficult to predict (PRE) and reuse (REU): filter adapter F (b); split-merge adapters S, M (c,d).

If ω_3 is a complex type, and A_4 only works on a *part of* the output ω_3, then an additional actor F must be added to the workflow (Fig. 5b) to filter the output of A_3 and so obtain the parts needed by A_4. Similarly, in Fig. 5c, suppose we wish to add actor A_{21} between two existing actors. A_{21} works only on specific parts of the output of A_2, and only produces a portion of the desired subsequent input type α_3. Here, we must add two new shim actors to satisfy the type constraints: (i) the split actor S separates the output of A_2 into the parts required by A_{21} and the remaining, "to-be-bypassed" parts; and (ii) the merge actor M combines the output of A_{21} with the remaining output of A_2, before passing on the aggregate to A_3. Finally, in Fig. 5d, a scientist might have discovered that she can optimize the workflow manually by replacing the actor A_2 with two specialized actors A_{21} and A_{22}, each working in parallel on distinct portions of the output of A_1. Similar to the previous case, this replacement requires the addition of two new shim actors to appropriately split and merge the stream. We note that it is often the case that a single workflow will require many of these "workarounds", not only making the workflow specification hard to comprehend, but also making it extremely difficult to construct in the first place.

In contrast, no shims are necessary to handle Fig. 5b–d in COMAD. In cases (b) and (c), actor configurations select relevant data items, passing everything else downstream. Similarly, (d) is implicitly and automatically achieved in COMAD simply by

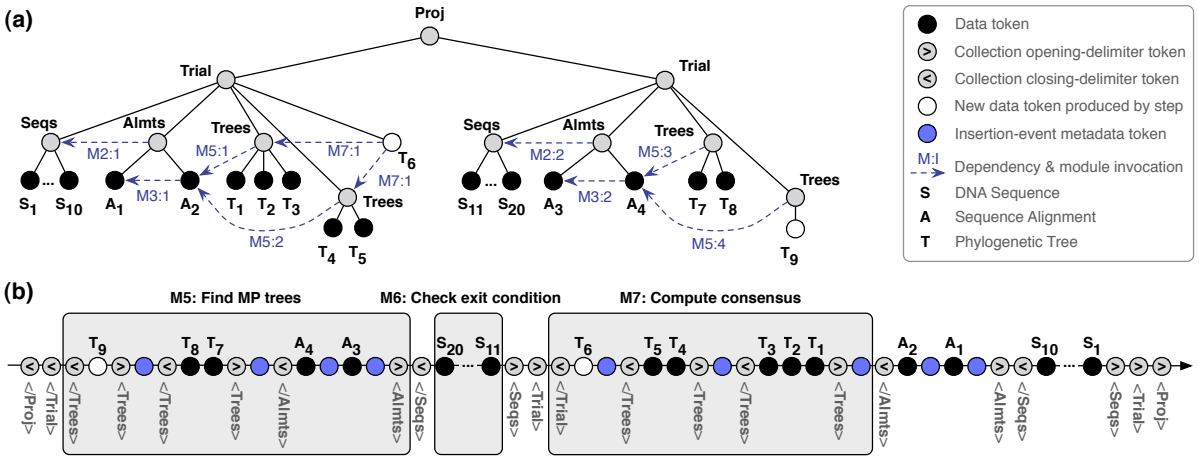

Fig. 3. An intermediate snapshot of a run of the COMAD phylogenetics workflow of Fig. 1: (a) the logical organization of data at an instant of time during the run; and (b) the tokenized version of the tree structure showing three modules (*i.e.*, actors) being invoked concurrently on different parts of the data stream. In COMAD, nested collections are used to organize and relate data objects that instantiate domain-specific types (*e.g.*, denoting DNA sequences S, alignments A, and phylogenetic trees T). A `Proj` collection containing two `Trial` sub-collections is used here to pipeline multiple sets of input sequences, and data products derived from them, through the workflow. In COMAD, provenance events for data and collection insertions, insertion dependencies, and deletions (from the stream) are added directly as metadata tokens to the stream (b), and can be used to induce provenance data-dependency graphs (a).

connecting A_{21} and A_{22} in series. Additionally in COMAD, the system can still optimize this to run A_{21} and A_{22} as task-parallel steps (described further below). In short, the use of this *part-of* subtyping in COMAD, based on configurations and scopes, enables more modular and change-resilient workflow designs than those developed using approaches based on strict (i.e., *is-a*) subtyping, since changes in irrelevant parts (e.g., outside the read-scope τ_α) will not affect the validity of the workflow design.

Due to the linear topology of assembly lines, COMAD workflows are also relatively easy to compose and understand. They resemble procedures such as recipes and lab protocols where the most important design criterion is that the specified sub-tasks be ordered to satisfy the dependencies of later tasks. For this reason, the meaning of a COMAD workflow often can be read directly from the workflow specification as in Fig. 1. Moreover, because most of the data manipulation and control flow constructs that typically clutter other workflows are not required in COMAD (the collection-management framework handles most of these tasks transparently), what is read off the workflow graph is the *scientific* meaning of the workflow.

Well-Formedness (WFV) via Type Propagation. A further benefit of requiring actors to declare read and write scopes is that we can employ *type inference* to determine various properties of COMAD workflows. The type inference problem for COMAD, denoted as

$$\tau \overset{\Delta_A}{\rightsquigarrow} \tau',$$

is to infer the modified schema $\tau' = \Delta_A(\tau)$ given an input type τ and an actor configuration Δ_A. We can restate the problem of finding τ' as

$$\tau' = (\underbrace{(\tau \ominus \tau_\alpha) \oplus \tau_\omega}_{=\Delta_A}),$$

which indicates that an actor configuration $\Delta_A : \tau_\alpha \rightarrow \tau_\omega$ can recognize parts τ_α of the input τ and *add* additional parts τ_ω (denoted by $\oplus$). It is also possible for the actor to *remove* the original τ_α parts from the stream (denoted in the formula by $\ominus$). If τ_α is not removed, we say that the actor is in "add-only" mode.

Using type inference, we can *propagate* inferred types downstream along any path

$$\tau \overset{\Delta_{A1}}{\rightsquigarrow} \tau_1 \overset{\Delta_{A2}}{\rightsquigarrow} \tau_2 \overset{\Delta_{A3}}{\rightsquigarrow} \cdots$$

once the initial input schema τ is known. Type propagation makes it possible to statically type-check (and thus *validate*) a COMAD workflow design. For example, if an actor's input constraint is violated, we say the actor A will *starve* (or is *extraneous*) for inputs of type τ. There can be different reasons why A can starve. In particular, either A's read-scope never matches anything in τ; or else, potential matches are not acceptable subtypes of τ_α. In both cases, the workflow can still be executed since the COMAD framework ensures that unmatched data simply flows through A unchanged. COMAD workflows are thus robust with respect to superfluous actors in a way that systems based on strict subtyping are not.

Predictability (PRE) via Type Propagation. Using static type inference, COMAD can help predict what a workflow will do when executed. Given an input schema and a workflow, we can compute the output schema of the workflow by propagating the schema information through the actors. Intermediate data products also can be inferred, together with information about which actors are used to create each product. Given an input schema (or collection structure), we can statically compute a *schema lineage graph*, which explains which actors (or analysis steps) refine and transform the input to finally produce the output.

The read and write scopes of actors in COMAD workflows also can be used to reveal inter-actor dependencies. In an assembly-line environment it is not a given that each worker uses the products introduced by the worker immediately up-

stream and no others. Similarly, an actor in a COMAD workflow might not work on the output of the immediately preceding coactor. Displaying to a workflow designer the actual dependencies would reveal accidently misconfigured actors that should be dependent on each other but are not due to scope mis-configurations, for example. Furthermore, we can statically infer the minimal data structure that must be supplied to a workflow such that all actors will find some data within their scope and so be invoked at least once during a run. COMAD thus allows us to provide scientists composing or examining workflows with a variety of predictions about the expected behavior of a workflow.

Optimization (OPT) via Pipeline Parallelism. In a manner similar to other dataflow process networks [LP95], actors in a COMAD workflow operate *concurrently* over items in the data stream. In COMAD, rather than supplying the entire tree-like structure of the data stream to each actor in turn, a sequence of tokens representing this tree is streamed through actors. For example, Fig. 3 illustrates the state of a COMAD run for the example workflow of Fig. 1 at a particular point in time, and contrasts the logical organization of the data flowing through the workflow in Fig. 3a with its tokenized realization at the same point in time in Fig. 3b. This figure further illustrates the pipelining capabilities of COMAD by including two independent sets of sequences in a single run. This degree of pipeline parallelism is achieved in part by representing nested data collections at runtime as "flat" token streams that contain paired opening and closing delimiters to denote collection membership.

Optimization (OPT) via Dataflow Analysis. Type propagation can also be used in COMAD workflows to minimize data shippings and maximize task parallelism. Consider the process pipeline

$$x \xrightarrow{\tau} \boxed{A} \longrightarrow \boxed{B} \longrightarrow \boxed{C} \xrightarrow{\tau'} y$$

denoted as $(A{\rightarrow}B{\rightarrow}C)$ for short, with input type τ and output type τ'. Type propagation starts with type τ and then applies actor configurations Δ_A, Δ_B, and Δ_C to determine, e.g., the parts of A's output (if any) that are needed as input to B and C. If, e.g., one or more data or collection items of A's output are not relevant for B and C (based on propagated type information), these items are automatically *bypassed* around actors B and C to y (or beyond, depending on the actors downstream of C). Thus, what looks like an otherwise linear workflow $(A{\rightarrow}B{\rightarrow}C)$ can be optimized using static type propagation and analysis. In this example, by "compiling" the linear workflow we might obtain one of the following process networks, based on the actual data dependencies of the workflow:

$$(A \shuffle B \shuffle C), \quad (A{\rightarrow}(B \shuffle C)), \quad ((A \shuffle B){\rightarrow}C)$$

where $(X \shuffle Y)$ denotes a task-parallel network with two branches, one for X and one for Y, respectively.

A simple example from physical assembly lines can further illustrate these optimizations. Consider a worker A who is operating on the front bumper (τ_A) of a car (τ). Other parts of the car (included in $\tau \ominus \tau_\alpha$) which are "behind" the bumper (in the stream) cannot move past A, despite the fact that they are irrelevant to A. In COMAD it is possible to optimize such a situation by "cutting up" the input stream and immediately bypassing irrelevant parts downstream (*e.g.*, to B or C). This minimizes data shipping costs and increases concurrency. In this case, we introduce into the network downstream *merge actors* that receive various parts from upstream *distribution actors*. Pairing of the correct data and collection items is done by creating so-called "holes"—empty nodes with specially assigned identifiers—and corresponding "filler" nodes [Zin08].

Recordability (REC) and Reportability (REP). We also illustrate in Fig. 3 how provenance information is captured and represented during a COMAD workflow run. As COMAD actors add new data and collections to the data stream, they also add special metadata tokens for representing provenance records. For example, the fact that Alignment2 (denoted A_2 in Fig. 3) was computed from Alignment1 (denoted A_1) is stored in the *insertion-event* metadata token immediately preceding the A_2 data token in Fig. 3b, and displayed as the dashed arrow from A_2 to A_1 in Fig. 3a. When items are not forwarded by an actor, *deletion-event* metadata tokens are inserted into the data stream, marking nodes as deleted so that they are ignored by downstream actors. From these events, it is possible to reconstruct and query data, collection, and process dependencies as well as determine the input and output data used for each actor invocation [BML08].

3.3. *Implementation of* COMAD

We have implemented many of the features of the COMAD framework described here and have included a subset of them in the standard Kepler distribution. [3] We also have employed COMAD as the primary model of computation in a customized distribution of Kepler developed for the systematics community. [4] The COMAD implementation in Kepler extends the PN (process network) director [LP95,MBL06,BL05], and provides a rich set of Java classes and interfaces for developing COMAD actors, managing and defining data types and collections, recording and managing runtime provenance events, and specifying coactor scopes and configurations.

We have developed numerous coactors as part of the COMAD framework and have used them to implement a variety of workflows. We have implemented actors for wrapping specific external applications, for executing web-based services, and for supporting generic operations on collections. We include tools in this framework for recording and managing provenance information associated with runs of COMAD workflows, including a generic provenance browser. To facilitate the reuse of conventional actors developed for use with Kepler, we provide as part of the framework support for conveniently wrapping SDF sub-workflows in a manner that allows them to be employed as Kepler coactors [MBL06].

[3] See `http://www.kepler-project.org`
[4] See `http://daks.ucdavis.edu/kepler-ppod`

To demonstrate the potential optimization benefits of COMAD, we also have recently developed a prototype implementation of a stand-alone COMAD workflow engine. The implementation is based on the Parallel Virtual Machine (PVM) library for message passing and job invocation, where each actor is executed as its own process and can run on a different compute node. Opening and closing delimeters (including holes and fillers) are sent using PVM messages; large data sets are managed as files on local filesystems and sent between nodes using secure copy (`scp`). Our experimental results have shown that the optimizations based on pipeline parallelism and dataflow analysis can lead to significant reductions in workflow execution time due to increased concurrency and fewer overall data shipments [Zin08]. As future work, we are interested in further developing this approach as part of the Kepler COMAD framework, allowing COMAD workflows designed within Kepler to be efficiently and transparently executed within distributed and high-performance computing environments.

3.4. *Limitations of* COMAD

Our applications of COMAD have shown that the advantages of this approach do come at a cost. First, COMAD workflows are easy to assemble only after the data associated with a particular domain has been modeled well. Until this is done, it can be unclear how best to organize collections of data passing through workflows, and challenging to configure coactor scope expressions (just as designing an assembly line for constructing an automobile would be difficult in the absence of blueprints and assembly instructions). On the other hand, once the data in a domain of research has been modeled well, this step need not be repeated again by others. COMAD makes it easy to take advantage of the data modeling work done by others, but it does not allow the data modeling step in workflow design to be skipped altogether.

Second, COMAD workflows cannot always be composed simply by stringing together a set of actors in an intuitive order. Often at least some of the coactors must be configured specifically for use in the context of the workflow being developed, and this requires an understanding of the assumed organization of data in the data sets to be provided as input to the workflow. We believe, however, that the design support tools described above will help make this step easier. Eventually, one can imagine workflow systems suggesting coactor configurations based on sample input data sets.

Third, many actors already have been developed for Kepler and other workflow systems, and these actors are not immediately useable as actors in COMAD workflows. As described above, however, we have developed an easy way to encapsulate conventional Kepler actors and sub-workflows within generic actors such that they can be employed seamlessly as coactors along with coactors originally developed as such.

Finally, while the assembly-line approach can make it easier for scientists to design and understand their workflows, a naïve implementation of a COMAD workflow enactment engine can result in a greater number of data transfers than would be expected for a more conventional workflow system. As discussed above, however, and described more fully in [ZBL07], static analysis of the scope expressions can be used to compile user-friendly, linear workflows into performance-optimized, non-linear workflows in which data is directly routed to just those actors that need it. Note that this optimization would be done at deployment or run time, leaving the workflow modeled by the scientist unchanged.

4. Related Work

The diversity of scientific data analyses requires that workflow systems address a broad range of complex issues. Numerous, equally diverse approaches have been proposed and developed to address each of these needs. The result is that there is no single, standard conceptual framework for understanding and comparing all of the contributions to this field, nor is there a common model for scientific workflow specifications shared across even a majority of the major tools. This situation is similar to that faced by the business workflow community [RtHEvdA05], where comparing the modeling support provided by systems based on Petri Nets, Event-Driven Process Chains, UML Activity Diagrams, and BPEL has proved challenging, and defining conceptual frameworks that are meaningful across all these approaches equally difficult.

In this paper, we have primarily focused on issues related to modeling and design of scientific workflows, a key area in which we believe much progress still remains to be made before scientists broadly adopt scientific workflow systems. In this section we relate this aspect of our work to modeling and design approaches reported by other groups. For a broad comparison of systems, we refer the reader to one of the many surveys on scientific workflow systems, e.g., [YB05].

COMAD is, indeed, one of many modeling and design frameworks for scientific workflows. Unlike other approaches, COMAD extends the process network (PN) dataflow model [LP95] by providing explicit support for nested collections of data, adding high-level actor scoping and configuration languages, and enabling implicit iteration of actors over (nested) collections of data. This paper extends our previous work [MBL06] on COMAD by (1) describing a set of general requirements that, if satisfied, would lead to wider adoption of workflow systems by scientists; (2) presenting the abstract modeling framework offered by COMAD in terms of virtual assembly lines and their advantages for workflow design; and (3) illustrating how COMAD satisfies the various design-oriented desiderata described above.

COMAD shares a number of characteristics with approaches for query processing over XML streams, e.g., [CCD+03,CDTW00,BBMS05,KSSS04,GGM+04,CDZ06]. Most of these approaches consider optimizations of specific XML query languages or language fragments, sometimes taking into account additional aspects of streaming data (e.g., sliding windows). COMAD differs by specifically targeting scientific workflow applications, by providing explicit support for modeling the flow of data through graphs of black-box

functions (actors), and by enabling pipeline and task-parallel concurrency without requiring the use of advanced techniques for preventing deadlocks and race conditions.

In common with [FPD+05,LAB+06,MSTW04], COMAD does *not* restrict workflow specifications to directed acyclic graphs (e.g., [OGA+06,DSS+05,BV04,CJSN03,BCS+05] do have this limitation). We have found that supporting advanced workflow modeling constructs such as loops; conditional branches; sub-workflows; nested, heterogeneous models of computation (e.g., composite coactors built from SDF subworklows); and so on, leads to specifications of complex scientific analyses that more clearly capture the scientific intent of the individual computational steps and of the overall workflow. The COMAD approach also can reduce the need for adapters and shims [HSL+04,OGA+06] through its virtual assembly-line metaphor, while still providing static typing support for workflows (e.g., as in [OGA+06,LAB+06]) via type progagation through read and write scopes. Taverna [OGA+06] provides a simple form of implicit iteration over intermediate collections, but without scope expressions and collection nesting; and the ASKALON system [FPD+05], provides management support for large collections of intermediate workflow data.

Finally, considerable work within the Grid community has focused on approaches for optimizing scientific workflows, with the aim of making it easy for users to specify, deploy, and monitor workflows, e.g., [FPD+05,TBF+06,BGK+07,BGH+03]. Our hope is that COMAD can leverage the automatic optimization techniques employed by these approaches, while providing scientists intuitive and powerful workflow modeling and design languages and support tools.

5. Conclusion

As a first step towards meeting the needs of scientists with little programming experience, we have identified and described eight broad areas in which we believe scientific workflow systems should provide modeling and design support: *well-formedness*, *clarity*, *predictability*, *recordability*, *reportability*, *reusability*, *scientific data modeling*, and *automatic optimization*, and have implemented a novel scientific workflow and data management framework that largely addresses these desiderata. While the goal of making it easy to develop arbitrary software applications might remain elusive forever, we believe that for scientific workflow automation there are good reasons for hope. We invite and encourage the community to join the quest for more scientist-friendly workflow modeling and design tools.

Acknowledgements. This work supported in part through NSF grants IIS-0630033, OCI-0722079, IIS-0612326, DBI-0533368, and DOE grant DE-FC02-07-ER25811.

6. References

[ABJF06] I. Altintas, O. Barney, and E. Jaeger-Frank. Provenance Collection Support in the Kepler Scientific Workflow System. In *Intl. Provenance and Annotation Workshop (IPAW)*, volume 4145 of *LNCS*. Springer, 2006.

[BBMS05] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *ACM SIGMOD*, 2005.

[BCS+05] L. Bavoil, S. P. Callahan, C. E. Scheidegger, H. T. Vo, P. Crossno, C. T. Silva, and J. Freire. VisTrails: Enabling Interactive Multiple-View Visualizations. In *IEEE Visualization*, page 18. IEEE Computer Society, 2005.

[BGH+03] A. Belloum, D. L. Groep, Z. W. Hendrikse, L. O. Hertzberger, V. Korkhov, C. T. A. M. de Laat, and D. Vasunin. VLAM-G: a grid-based virtual laboratory. *Future Generation Comp. Syst.*, 19(2):209–217, 2003.

[BGK+07] M. Bubak, T. Gubala, M. Kasztelnik, M. Malawski, P. Nowakowski, and P. Sloot. Collaborative virtual laboratory for e-Health. In P. Cunningham and M. Cunningham, editors, *Expanding the Knowledge Economy: Issues, Applications, Case Studies*. IOS Press, 2007.

[BL05] S. Bowers and B. Ludäscher. Actor-Oriented Design of Scientific Workflows. In *Intl. Conference on Conceptual Modeling (ER)*, LNCS. Springer, 2005.

[BLNC06] S. Bowers, B. Ludäscher, A. H. Ngu, and T. Critchlow. Enabling Scientific Workflow Reuse through Structured Composition of Dataflow and Control-Flow. In *Post-ICDE Workshop on Workflow and Data Flow for Scientific Applications (SciFlow)*, 2006.

[BML08] S. Bowers, T. M. McPhillips, and B. Ludscher. Provenance in Collection-Oriented Scientific Workflows. *Concurrency and Computation: Practice and Experience*, 20(5):519–529, 2008.

[BV04] R. Buyya and S. Venugopal. The Gridbus toolkit for service oriented grid and utility computing: an overview and status report. In *Intl. Workshop on Grid Economics and Business Models (GECON)*, 2004.

[CCD+03] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR'03)*, 2003.

[CDTW00] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagraCQ: A scalable continuous query system for internet databases. In *ACM SIGMOD*, pp. 379–390, 2000.

[CDZ06] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. In *Intl. Conf. on Data Engineering (ICDE)*, 2006.

[CJSN03] J. Cao, S. Jarvis, S. Saini, and G. Nudd. GridFlow: Workflow Management for Grid Computing. In *Intl. Symp. on Cluster Computing and the Grid (CCGrid)*, 2003.

[DRGS07] D. De Roure, C. Goble, and R. Stevens. Designing the myExperiment Virtual Research Environment for the Social Sharing of Workflows. In *IEEE Intl. Conf. on e-Science and Grid Computing*, pp. 603–610, 2007.

[DSS+05] E. Deelman, G. Singh, M.-H. Su, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, K. Vahi, G. Berriman, J. Good, A. Laity, J. C. Jacob, and D. S. Katz. Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. *Scientific Programming Journal*, 13(3):219–237, 2005.

[FPD+05] 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. In *IEEE Grid Computing Workshop*, 2005.

[GDE+07] Y. Gil, E. Deelman, M. Ellisman, T. Fahringer, G. Fox, D. Gannon, C. Goble, M. Livny, L. Moreau, and J. Myers.

Examining the Challenges of Scientific Workflows. *IEEE Computer*, 40(2):24–32, 2007.

[GGM+04] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Transactions on Database Systems (TODS)*, 29(4):752–788, 2004.

[GLM04] P. Groth, M. Luck, and L. Moreau. A protocol for recording provenance in service-oriented grids. In *Intl. Conf. on Principles of Distributed Systems*, 2004.

[GRD+07] Y. Gil, V. Ratnakar, E. Deelman, G. Mehta, and J. Kim. Wings for Pegasus: Creating Large-Scale Scientific Applications Using Semantic Representations of Computational Workflows. In *Proc. of the AAAI Conference on Artificial Intelligence*, pp. 1767–1774, 2007.

[HK03] S. Hwang and C. Kesselman. GridWorkflow: A Flexible Failure Handling Framework for the Grid. In *IEEE Intl. Symp on High-Performance Distributed Computing (HPDC)*, pp. 126–137, 2003.

[HSL+04] D. Hull, R. Stevens, P. Lord, C. Wroe, and C. Goble. Treating shimantic web syndrome with ontologies. In *First Advanced Knowledge Technologies Workshop on Semantic Web Services (AKT-SWS04)*, 2004.

[JHM04] W. M. Johnston, J. P. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, 204.

[KSSS04] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based Scheduling of Event Processors and Buffer Minimization for Queries on Structured Data Streams. In *VLDB Conf.*, 2004.

[LAB+06] 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*, pp. 1039–1065, 2006.

[LP95] E. A. Lee and T. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.

[LSV98] E. A. Lee and A. L. Sangiovanni-Vincentelli. A framework for comparing models of computation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1217–1229, 1998.

[MBL06] T. McPhillips, S. Bowers, and B. Ludscher. Collection-Oriented Scientific Workflows for Integrating and Analyzing Biological Data. *3rd International Workshop on Data Integration in the Life Sciences (DILS'06)*, 2006.

[ML08] L. Moreau and B. Ludäscher, editors. *Concurrency and Comptuation: Practice and Experience, Special Issue on The First Provenance Challenge*, volume 20(5). Wiley, 2008.

[Mor94] J. P. Morrison. *Flow-Based Programming – A New Approach to Application Development*. Van Nostrand Reinhold, 1994.

[MSM97] D. Maddison, D. Swofford, and W. Maddison. NEXUS: An extensible file format for systematic information. *Systematic Biology*, 46(4):590–621, 1997.

[MSTW04] S. Majithia, M. S. Shields, I. J. Taylor, and I. Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. In *ICWS*, pp. 514–. IEEE Computer Society, 2004.

[OGA+06] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: Lessons in Creating a Workflow Environment for the Life Sciences. *Concurrency and Computation: Practice & Experience*, pp. 1067–1100, 2006.

[PA06] C. Pautasso and G. Alonso. Parallel computing patterns for grid workflows. In *Workshop on Workflows in Support of Large-Scale Science (WORKS)*, 2006.

[RLSR+06] U. Radetzki, U. Leser, S. C. Schulze-Rauschenbach, J. Zimmermann, J. Lüssem, T. Bode, and A. B. Cremers. Adapters, shims, and glue—service interoperability for in silico experiments. *Bioinformatics*, 22(9):1137–1143, 2006.

[RtHEvdA05] N. Russell, A. ter Hofstede, D. Edmond, and W. van der Aalst. Workflow data patterns: identification, representation and tool support. In *Conf. on Conceptual Modeling (ER)*, volume 3716 of *LNCS*, pp. 353–368, 2005.

[SdSGS06] L. Salayandia, P. P. da Silva, A. Q. Gates, and F. Salcedo. Workflow-Driven Ontologies: An Earth Sciences Case Study. In *Intl. Conf. on e-Science and Grid Technologies (e-Science)*, page 17, 2006.

[SPG05] Y. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Record*, 34(3):31–36, 2005.

[TBF+06] H. Truong, P. Brunner, T. Fahringer, F. Nerieri, and R. Samborski. K-WfGrid distributed monitoring and performance analysis services for workflows in the grid. In *IEEE Conf. on e-Science and Grid Computing (e-Science)*, 2006.

[THG94] J. Thompson, D. Higgins, and T. Gibson. CLUSTAL W: improving the sensitivity of progressive multiple sequence alignments through sequence weighting, position specific gap penalties and weight matrix choice. *Nucl. Acids Res.*, 22:4673–2680, 1994.

[TTK+07] T. Tavares, G. Teodoro, T. Kurc, R. Ferreira, D. Guedes, W. Meira Jr., U. Catalyurek, S. Hastings, S. Oster, S. Langella, and J. Saltz. An efficient and reliable scientific workflow system. In *Intl. Symp. on Cluster Computing and the Grid (CCGrid)*, 2007.

[WGG+07] C. Wroe, C. A. Goble, A. Goderis, P. W. Lord, S. Miles, J. Papay, P. Alper, and L. Moreau. Recycling workflows and services through discovery and reuse. *Concurrency and Computation: Practice and Experience*, 19(2):181–194, 2007.

[YB05] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Record*, 34(3):44–49, 2005.

[ZBL07] D. Zinn, S. Bowers, and B. Ludscher. Change-Resilient Design and Dataflow Optimization for Distributed XML Stream Processors. Technical Report CSE-2007-37, UC Davis, 2007.

[Zin08] D. Zinn. Modeling and optimization of scientific workflows. In *Proc. of the EDBT Ph.D. Workshop*, 2008.

[ZWF06] Y. Zhao, M. Wilde, and I. Foster. Applying the Virtual Data Provenance Model. In *Intl. Provenance and Annotation Workshop (IPAW)*, volume 4145 of *LNCS*. Springer, 2006.