Scientific Workflow Design with Data Assembly Lines

Daniel Zinn [*] Shawn Bowers [†] Timothy McPhillips [‡] Bertram Ludäscher [*,‡]

[*]Department of Computer Science, University of California, Davis
[†]Department of Computer Science, Gonzaga University
[‡]UC Davis Genome Center
{dzinn,sbowers,tmcphillips,ludaesch}@ucdavis.edu

ABSTRACT

Despite an increasing interest in scientific workflow technologies in recent years, workflow design remains a challenging, slow, and often error-prone process, thus limiting the speed of further adoption of scientific workflows. Based on practical experience with data-driven workflows, we identify and illustrate a number of recurring scientific workflow design challenges, i.e., parameter-rich functions; data assembly, disassembly, and cohesion; conditional execution; iteration; and, more generally, workflow evolution. In conventional approaches, such challenges usually lead to the introduction of different types of "shims", i.e., intermediary workflow steps that act as adapters between otherwise incorrectly wired components. However, relying heavily on the use of shims leads to brittle (i.e., change-intolerant) workflow designs that are hard to comprehend and maintain. To this end, we present a general workflow design paradigm called *virtual data assembly lines* (VDAL). In this paper, we show how the VDAL approach can overcome common scientific workflow design challenges and improve workflow designs by exploiting (i) a semistructured, nested data model like XML, (ii) a flexible, statically analyzable configuration mechanism (e.g., an XQuery fragment), and (iii) an underlying virtual assembly line model that is resilient to workflow and data changes. The approach has been implemented as Kepler/COMAD, and applied to improve the design of complex, real-world workflows.

1. INTRODUCTION

Scientific workflows are increasingly used for the integration of pre-existing tools and algorithms to form larger, more complex applications, e.g., for scientific data analysis, computational science experiments and the associated simulation management [33, 11, 21]. In this sense, scientific workflow development resembles shell scripting rather than general purpose programming, i.e., specialized programs (implemented, e.g., in C++, Fortran, Java or with custom tools such as Matlab, R, etc.) are only "wrapped" and chained together rather than fully programmed in the workflow system itself. This is similar to the efforts towards a Common Component Architecture (CCA) for high-performance scientific computing [2], in which specialized programs are invoked as separate processes and chained together, rather than compiled together into a single executable, thus allowing integration of components from third parties and implemented in different languages. Compared to script-based approaches to automation, scientific workflow systems promise to offer a number of advantages, e.g., built-in support for recording and querying data provenance [9], for deploying workflows on cluster or Grid environments [10], for parameter sweeps [1], and for dataflow and concurrency optimization [38, 36], to name a few. However, arguably one of the biggest opportunities—and challenges—lies in the promise to be useable by "mere mortals" [25]: Scientists who are neither experts in scripting languages nor in (distributed) software development, should be able to develop and maintain understandable workflows. Ideally, scientific workflow design should feel less like (visual) programming or scripting, and more like story-telling. For example, a scientist should be able to assemble pre-existing software components into simple, mostly linear dataflow pipelines, such that consecutive steps, when read out loud, correspond to high-level descriptions of a computational protocol or e-Science experiment.

All too often, however, current scientific workflow designs include overly complex wiring structures and are cluttered with various forms of software *shims*[1] (adapters) to align or mediate mismatching workflow steps [16]. For example, [19] state that a recent study of 560 scientific workflows from the myExperiment repository [29] showed that over 30% of workflow tasks are shims. Shims arise frequently when the output produced by a workflow step is not compatible with the inputs that a subsequent step can consume, which in turn is common when combining independently developed services or functions into larger workflows. For example, [16] report that bioinformatics services suffer from "shimantic web syndrome": many services are *nearly* compatible (it makes sense to chain them together), yet they require intermediate shims before the service chains can be executed. Similarly, *complex wiring* can contribute to "messy" designs [20].

The proliferation of scientific workflows with many shims and complex wiring is a limiting factor to their wider adoption and use. For example, such designs are hard to understand (shims distract from "the real story", i.e., the experiment protocol) and difficult to maintain (custom shims and wiring make the design brittle and sensitive to changes in the data structures and workflow steps). Among other things, overly complex or obfuscated designs limit the use of workflows for documenting and communicating the underlying ideas of the implemented scientific method, require workflow experts (similar to script programmers), and increase the cost for workflow development and maintenance. Even if tools provide a clear visual distinction between shims and scientifically relevant steps, workflow designers still need to place these components and

[1]a (physical) shim is a thin strip of metal for aligning pipes

connect them. Taverna [32], e.g., allows scientists to declare certain steps as "boring", which hides them from the canvas, thus visually simplifying the design. Nevertheless, changes to the workflow (e.g., adding new steps) still require connecting hidden shims with other visible or invisible steps.

Contributions. Based on practical experiences with the design of data-driven workflows from various domains (e.g., see [21]), we first identify a number of workflow design challenges and illustrate them with examples and use cases (Section 3). Specifically, we elaborate on the challenges resulting from parameter-rich functions; data assembly/disassembly and data cohesion; conditional execution; iteration; and, more generally, workflow evolution. We then present a general workflow design paradigm called *virtual data assembly lines* (VDAL) which has been implemented as Kepler/COMAD [24], but which is applicable to other data-driven systems (e.g., Taverna or Triana [34]) as well (Section 4). The crux of VDAL is that shims and complex wiring are minimized by encapsulating conventional black-box functions and services inside of a "data selection and shimming" layer that can be manually or automatically configured. We describe in detail the anatomy of VDAL components, i.e., the signatures and effects of operations inside of such components for scoping, binding, iterating over, and placing data. Finally, we show how our approach addresses the workflow design challenges mentioned above (Section 5). We summarize our results and discuss other related work in Section 6.

Relation to Prior Work. Hull et al. [16] highlight and classify some of the problems relating to shims, and propose to treat them by employing ontologies as semantic types. An approach to infer data transformation shims using semantic annotations is described in [3]. The authors of [14] emphasize the use of semantic representations and planning in scientific workflows. Arguably the work closest to ours with respect to the goal of solving the shimming problem is [19] which focuses on type mismatches between the data types of consecutive steps and calls this a shimming problem of "Type I"; the authors describe as "Type II" the problem of mismatching connections of a scientific task that is nested within another, enclosing component that wraps the inner task. Our results extend [19] (e.g., we consider additional design challenges, not just Type I and II problems), and are complementary, e.g., to [16, 4, 14] (e.g., we do not consider semantic mismatches). In [25] some general advantages of a collection-oriented, assembly-line style design were presented, but no concrete examples were discussed. [37] provides more details about how VDAL designs can be implemented via an existing XML-processing language and shows how to provide additional features for the workflow designer via static analysis. In [38] we presented another instance of VDAL called Δ-XML that optimizes data shipping in distributed workflow settings, and we showed how simple, linear designs can be automatically translated into optimized (but wiring-intensive) low-level designs.

2. PRELIMINARIES

In the following, we introduce our terminology and basic notions. A *scientific workflow* is a description of a process, usually in terms of scientific computations and their dependencies [22], and can be visualized as a directed graph, whose nodes (also called *actors*) represent workflow steps or tasks, and whose edges represent dataflow and/or control-flow [11]. A basic formalism is to use directed, acyclic graphs (DAGs), where an edge $A \rightarrow B$ means that actor B can start only after A finishes (a control-flow dependency). With such a DAG-based workflow model (e.g., used by Condor/DAGMan [8]) one can easily capture serial and task-parallel execution of workflows, but other data-driven computations, e.g., data streaming, pipeline-parallelism, and loops cannot be represented directly in this model. In contrast, dataflow-oriented computation models, like those based on Kahn's *process networks* [17], in addition support pipeline-parallel computation over data streams, and cyclic graphs to explicitly model loops. This model underlies most Kepler workflows, and *mutatis mutandis*, applies to other scientific workflow systems with data-driven models of computation as well (e.g., Taverna [32], Triana [34], Vistrails [13], etc.) In these data-flow oriented workflow models, each edge represents a unidirectional *channel*, which connects an *output port* of an actor to an *input port* of another actor. Channels can be thought of as unbounded queues (FIFO buffers) that transport and buffer *tokens* that flow from the token-producing output port to the token-consuming input port. For workflow modeling and design purposes, it makes sense to distinguish different kinds of ports: A *data port* (the default) is used by an actor A to consume (read) or produce (write) data tokens during each *invocation* (or *firing*) of A. In contrast, a *control port* of A is a special input port whose (control) token value is not used by A's invocation, but which can trigger A's execution in the first place. An actor *parameter* can be seen as a special, more "static" input port from which data usually is not consumed upon each invocation, but rather remains largely fixed (except during parameter sweeps). While actor data ports are used to stream data in and out of an actor, actor parameters are typically used to configure actor behavior, set up connection or authentication information for remote resources, etc. A *composite actor* encapsulates a subworkflow and allows the nested workflow to be used as if it were an atomic actor with its own ports and parameters.

While the Kahn model is silent on the data types of tokens flowing between actors, practical systems often employ a structured data model. However, in practice, when actors implement web service calls or external shell commands, data on the wire is often of type `string` or `file`, even if the data conceptually has more structure. Kepler natively employs a model with structured types (inherited from Ptolemy [7]), including records and arrays. When sending data from one actor to another, this creates many options. For example, a list of data can be sent from one actor to another in a single *array token*, or as a stream of tokens corresponding to the elements of the list. Similarly, large *record tokens* can be assembled and later broken into smaller fragments. These choices can in fact complicate workflow design (see below), whereas the proper use of a serializable, semistructured model such as XML allows a more flexible and uniform data treatment. VDAL and its instances Kepler/COMAD and Δ-XML employ such a model.

3. WORKFLOW DESIGN CHALLENGES

Here we identify and describe common scientific workflow use-cases and design challenges we have encountered in practice when applying a conventional dataflow modeling approach. We revisit these challenges in Section 5 and show how VDAL addresses them.

3.1 Parameter-Rich Functions and Services

Many scientific functions have a large number of input ports and parameters. For example, DNAML (DNA Maximum Likelihood) from PHYLIP (the Phylogeny Inference Package [12]) takes 10 parameters in addition to the list of DNA sequences it operates on. In the conventional modeling approach, actors that wrap such applications have many ports, each connected to a distinct channel. This quickly leads to complex wiring when several of these components are used. One current solution is to use actor parameters for specifying the input values that do not change during workflow execution. However, this approach leads to less flexible actors because once an input has been modeled as a conventional parameter,

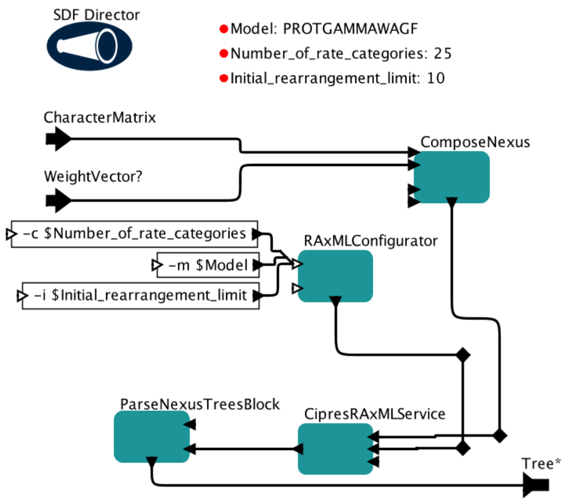


Figure 1: Parameter-rich service, record assembly and disassembly. A Cipres RAxML service is used to infer phylogenetic trees from aligned protein sequences, provided in `CharacterMatrix`. Besides the scientifically meaningful actor `CipresRAxMLService`, there are 6 auxiliary actors: `ComposeNexus`, `RAxMLConfigurator`, and three String actors are used to assemble the input data; `ParseNexusTreesBlock` is used to disassemble and convert the result for subsequent steps.

its value cannot be changed at workflow run time.

Example: CipresRAxML. The `CipresRAxML` composite actor in the Kepler/pPOD release [6] expects a `CharacterMatrix` containing DNA sequences and optionally a `WeightVector` as inputs. Furthermore, values for the parameters `rate_categories`, `model`, and initial rearrangement limit have to be specified. Figure 1 shows a screenshot of the *inside* of the `CipresRAxML` composite actor. The composite uses three *parameters* to configure the RAxML service in addition to the two data input ports. While this approach successfully reduces the number of input ports to two, it also reduces flexibility. Since parameter values are fixed during workflow execution, the composite actor is not re-usable in a workflow that, e.g., iterates over the `model` parameter to find an optimal setting. Here, the actor would need to be modified to have an additional input port for the `model` input. A `CipresRAxML` actor that allows all its configuration to be changed during a workflow run would need five input ports.[2] Although some of the parameters are for optimizing execution only, most of them do in fact influence the scientific result of the workflow. The `CipresRAxMLService` actor included in the Kepler/pPOD release only makes three of the parameters visible, restricting the service's flexibility even more. However, even the use of five distinct ports quickly becomes a modeling problem if the service should be called more than once. A full set of inputs needs to be sent for each invocation, and this quickly results in overly complex designs when loops or conditional execution are involved (as described below).

Packaging Inputs and Outputs. To address the problem of having too many ports, components sometimes expect all input data bundled together in a single large record data structure and similarly output all data produced during a single invocation bundled in a new record. A prominent example is the typical document-style web service that expects one large SOAP message and returns another SOAP message. Although these components have but a single input and a single output port, the general problem is not solved: complex input records must be assembled and outputs must be disassembled, tasks usually performed via shim-actors which themselves necessarily have many ports. Consequently, complex wiring still occurs between these shims. Some scientific workflow tools support automatic creation of shims for web services, based on the operation's WSDL specification. While such features clearly help the workflow designer, the shim actors still need to be connected, and can clutter the workflow. The problem has been recognized, e.g., by the Taverna developers; Taverna provides a feature to hide these shims via a "boring" tag. However, the underlying problem remains: new components added to a workflow still must be connected with the hidden shims to work properly.

The subworkflow inside the `RAxML` composite actor in Fig. 1 uses two different types of shims for record assembly: (1) `ComposeNexus` is an example of a *black-box required record*; Nexus is a textual container format that can contain different data such as a character matrix, a weight vector, and sequence data. The `RAxML` service expects its input data in this specific format, and the designer of the `RAxML` actor thus has to assemble such a format. (2) The second kind of shim is not required by the given black-box function. Instead an *ad hoc record* is employed by the workflow designer: here, e.g., `RAxMLConfigurator`, is used to create a custom configuration record to bundle all configuration information into a single token. In the following sections, we will show more examples of such ad hoc record management. While records required by black-boxes must be constructed at some point in the process, our approach completely eliminates shims arising from ad hoc data records.

3.2 Maintaining Data Cohesion

Individual data items processed by scientific workflows often are related to each other in important ways. When DNA, RNA, or protein sequences are aligned, for example, multiple possible alignments often are computed, each of which can have various quality assessment scores. In an automated version of a workflow computing and comparing multiple alignments, the system must maintain relationships between parts of the input data sets and portions of the workflow output. In order to maintain this *data cohesion*, current designs often create *ad hoc* records and array tokens. This approach has immediate drawbacks: (1) Packaging large amounts of data into one array often reduces workflow concurrency. When a large array token is created from an incoming stream of data, earlier arriving data is not sent to the next actor until the whole array is assembled. (2) Records and arrays have to be assembled and disassembled, which leads to shims that clutter the workflow graph and easily lead to complex routings. (3) Workflow designs are not very adaptive to changes in the data organization. Consider a record that contains an alignment and a score value. A subworkflow that requires such a record cannot easily process an array of alignments—even if the array contains the records as elements.

Example: Record Management. The workflow in Fig. 2 uses a specific R model for a bioinformatics sequencing task [31]. A single array token input at the upper left corner is disassembled into individual element tokens, which are then used to build a custom record. The single record is routed to four record disassemblers that provide the raw data to four `R` actors. Once the subnetwork of `R` actors is run, a final genotype record is assembled and sent to the output port. Clearly, such low-level data access and repacking operations distract from the primary functions of the workflow.

Example: Data associations. Figure 3(a) shows four hypothetical actors for genomics research, each of which takes a single data object and creates a list of related results. `BLAST` is used to find similar DNA sequences to a given DNA sequence; `MOTIFSearch` is used to detect one or more motifs, i.e., repeated patterns, in a DNA se-

[2]RAxML [30] by itself has 32 command-line parameters!

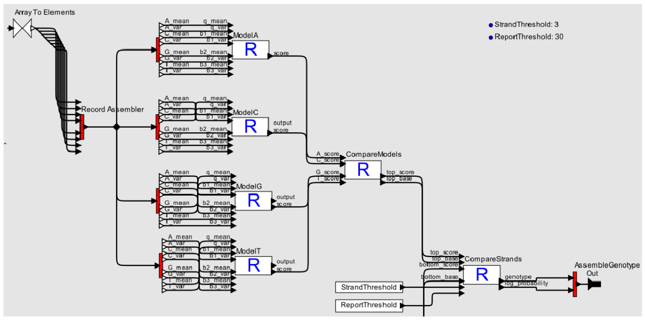


Figure 2: Record-handling shims. Record disassemblers and assemblers are used to unpack, restructure and repack data in [31].

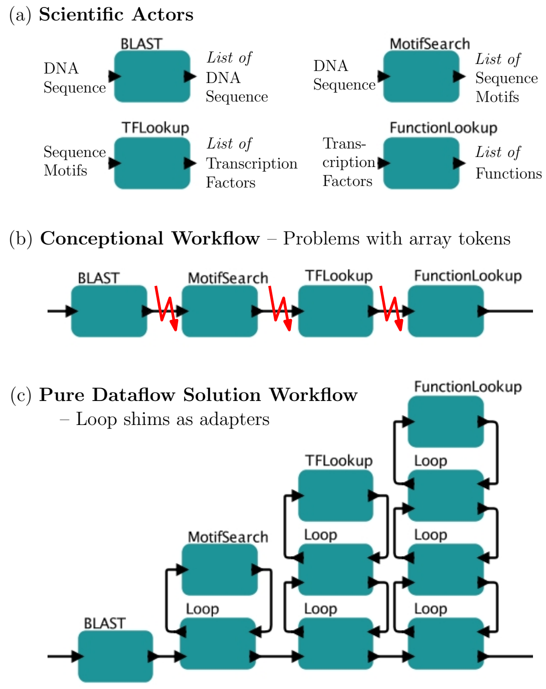


Figure 3: Data cohesion. Maintaining nesting structure using array tokens and additional loops.

quence; and from a given sequence motif `TFLookup` searches for proteins (transcription factors) reported to bind to the motif; finally via `FunctionLookup`, the specific biological functions associated with a transcription factor can be obtained. As shown in Fig. 3(b), we ideally would to be able to chain some or all of these actors together, e.g. to predict which functions a particular input DNA sequence is potentially associated with. However, there are problems with this approach: If each of the actors just outputs a token list for each received input token, then the associations between the data is lost. Consider a workflow input that has two DNA sequences, for each of which `BLAST` will return a list of similar sequences. However, on the output side of the `BLAST` actor, the two lists are not distinguishable from each other any longer, i.e., the output sequences are not grouped into two lists.

One solution to maintain the groupings is to use an array as output structure at each service. However, if we then want to chain the actor `MotifSearch` to analyze the results of a `BLAST` search for a given DNA sequence, then we need to insert a special looping shim that unpacks the array and sends it one-element-after-another to `MotifSearch`, as shown in Fig. 3(c). Then, `MotifSearch`'s returned array of motifs are packaged together to form an array of arrays of motifs. To extend the pipeline further, the actor `TFLookup` would need two of these looping shims to work properly. Furthermore, consider the case in which we would like to use multiple arrays of DNA sequences as overall workflow input. Here, we would need to add additional `Loop` shims around all existing actors as shown in Fig. 3(c).

A solution for this problem is provided by Oinn [27] and implemented in the Taverna workflow system: The workflow system itself is made aware of array-structures, and the system itself automatically inserts these looping constructs if there are type mismatches. The Loop (or `map`) operations are not made explicit as actors in the system and thus workflows are kept clean, and most importantly, easy to evolve. Our solution is similar to Taverna's, insofar that we also enrich the simple data model of dataflow networks. However, instead of (nested) lists, we will use labeled (nested) lists with annotations, a data model that corresponds to XML. We can therefore additionally use XML techniques to select the desired data to be fetched from the actor's the input.

3.3 Conditional Execution

Conditional executions and data filtering steps are essential in many scientific analyses. Consider, for example a workflow that infers phylogenetic trees from a set of input sequences, and then computes a consensus tree only from these result trees that fulfill certain quality criteria, e.g., that have a strong support. Here, only trees with high support should be used as input to the consensus step.

In current models, many control-flow constructs are encoded into the dataflow network, which leads to workflows with many shim actors and complex wiring [5]. For example, an If-Then-Else like filter can be mapped into a dataflow graph as a control-flow actor and two distinct routes. Such control-flow actors together with their necessary wiring lead to complex designs for moderately-sized workflows [28].

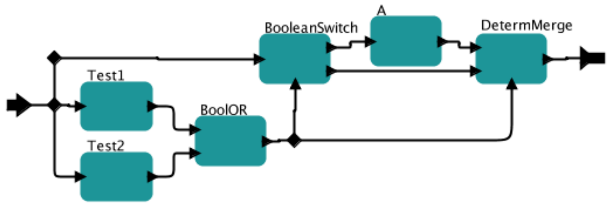


Figure 4: Dataflow model for "if (Test1 or Test2) then A"

Example: If-Then-Else. Consider an example workflow in which an analysis `A` should be applied to incoming data only if at least one of two tests (performed via the actors `Test1` and `Test2`) is successful. A common way to achieve this goal (see Fig. 4) is to use the actors `Test1` and `Test2`, route their Boolean output to an `OR` that combines the truth-values, which is then routed to the control input of a `BooleanSwitch`. The switch will then route the data either to `A` or to a following `Merge` actor, which combines the two streams of data to one. To maintain the order of data tokens, the switch sends a control signal to the merge. Depending on this signal, the merge is reading data from port 1 or 2, respectively.

Note that this solution does not only deploy three additional shim actors (or, switch, merge) with intricate wiring, but it will even fail working properly if actor `A` outputs more then one token for each token read on the input port.

Note that the workflow in Fig. 4 is a very small example, which only implements the simple control-flow *if (Test1 or Test2) then A*. In practice, many more of these conditional executions might be used, which quickly lead to even more complex designs with many shims [5].

3.4 Iterations over Cross Products

Scientific workflows can be used to automate analyses over multiple, independent input data sets; repeat analyses on the same data set with multiple sets of parameter values; or both. Combining data and parameter sweeps resembles executing the analysis over a cross-product of some subset of the available data. In conventional dataflow approaches these cross-products have to be constructed manually via actors that loop over sets of data and iteratively output the data as input to a downstream actor.

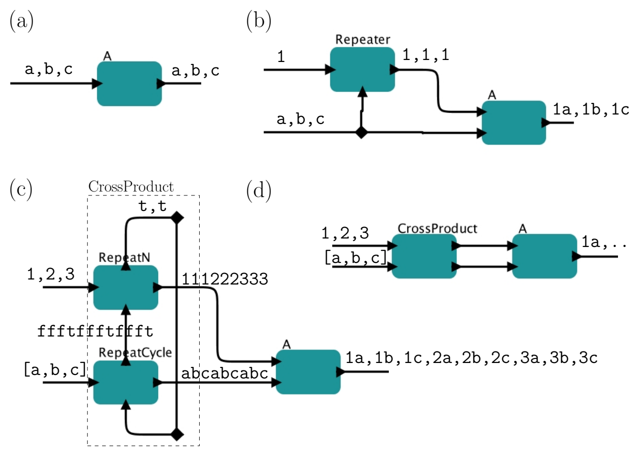


Figure 5: Conventional cross-products. Low-level models explicitly compute the cross-product. Tokens are given in "reading order", i.e., `1,2,3` means first token `1`, then `2`, and `3` are sent.

Example: Cross-products. Figure 5 shows workflows that invoke an actor `A` iteratively over combinations of incoming data items. When the A actor has only one input port, see Fig. 5(a), the data can be provided directly on the input channel. Here, the FIFO queue semantic of dataflow networks achieves the desired result of multiple invocations of A with the different input data `a`, `b`, and `c`. Now, consider an actor `A` with more than one input port as in Fig. 5(b). If we want to iteratively execute `A` over a list of inputs to one port while keeping the input to another port constant, then an additional `Repeater` actor is needed to explicitly construct the cross-product of the input data (`1`) × (`a`, `b`, `c`) that is routed to `A`. First, the `Repeater` reads the input data `1` into internal memory. Then, upon receiving a signal on the trigger port at the bottom of the actor, it will output a copy of the stored data. Now consider the case of two lists of inputs (`1`, `2`, `3`) and (`a`, `b`, `c`) where we want to invoke `A` on each element of their cross-product. Now even more control-flow actors and routes are necessary: Figure 5(c) shows a feasible design involving two specialized repeat actors wired together in a complex, crisscrossed pattern that takes considerable thought to design (and even more work to explain to others). Note, that this design works in a streaming fashion on the upper input but not on the lower one, i.e., the array `[a,b,c]` is consumed completely before any data is output, whereas the data on the above channel can be provided incrementally.

Larger workflow designs are necessary if more than two lists are involved in the cross-product. The design of the cross-product-generating actors could be placed into a composite actor as shown in Fig. 5(d) to hide the details from the modeler. However, different versions of this actor would need to be created for (i) different numbers of input ports, (ii) to realize other features such as streaming, and (iii) to accommodate input in single arrays or as streams.

Loops over input data as shown in the previous examples are very common in scientific workflows, e.g. for multiple-dimension parameter sweeps. Therefore, the workflow system should make it easy for the workflow designer to construct these loops without worrying about the low-level details of data buffering and routing. Our approach will provide a declarative way of specifying these loops. Furthermore, since our data is organized in nested collections, explicit distinction between array and non-array tokens is not necessary. Thus, the user can concentrate on specifying *what* data should be involved in cross-products, and the workflow system itself can choose *how* to compute them.

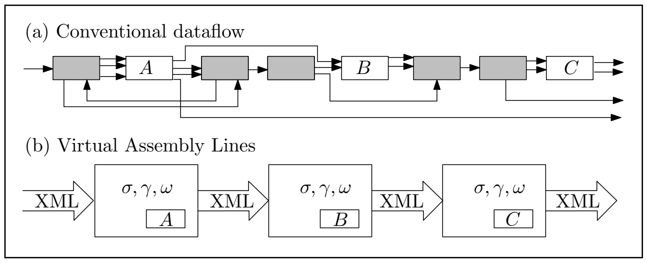


Figure 6: Conventional vs. VDAL. In VDAL, data massaging is moved to a configuration layer (σ, γ, ω) with declarative, local shims, minimal wiring complexity, thus facilitating simpler designs and reusable actors.

3.5 Workflow Evolution

In real-world workflows, the sorts of use cases and design patterns described above are interwoven in intricate ways, making workflows not only hard to understand, but also hard to modify and evolve. Figure 6(a) schematically shows a dataflow network comprising three scientifically meaningful actors (`A`, `B`, and `C`), and a number of shim actors (depicted as grey boxes). To insert a new scientific actor into such a network, it is necessary to understand the wiring and complex interactions between existing control-flow shims.

In contrast, a virtual data assembly line Fig. 6(b) localizes control-flow inside well-structured, configurable shells around black-box actors. Data is not broken up and scattered across different wires, but flows as XML-like structures from one VDAL actor to the next. Each such actor can locally determine which portions of the incoming data stream are of interest and which can be ignored and passed unprocessed to downstream actors. To insert an actor into a VDAL workflow, the workflow creator needs to know primarily the XML schema on the stream between the actors. These schemas often correspond to folder structures or scientifically meaningful hierarchies, and can thus be very intuitive for the modeler [24]. One can also co-design schema and workflow while utilizing automatic schema propagation through already configured actors [38].

4. VIRTUAL DATA ASSEMBLY LINES

We combine ideas from process networks with XML queries, updates, and stream processing. Specifically, our approach, *Virtual Data Assembly Lines* (VDALs) is characterized by the following three ideas (see Fig. 6): *(1) Linear Workflows:* A data assembly line always contains a linear workflow graph. That is, each actor has exactly one input and one output port. *(2) Structure-rich channels:* The data flowing from port to port is structured as labeled trees possibly with additional attributes much like XML data. The data is streamed in a SAX-like manner on the channels, although different execution strategies are possible [38, 36]. This is in contrast to common approaches where data on channels is of simple types or custom-made record and array types. *(3) Configuration shell:* Scientific components are wrapped in a "white-box" data selection and shimming layer which scientists can configure to specify what input data is taken from the input stream and where the result of the components application is put back into the stream. Here, we devise a domain-specific language to minimize and localize shimming tasks, e.g., those tasks performed by record-assembler and disassembler shims in conventional approaches. Moving the data selection and shimming into a configurable layer around each actor not only reduces the wiring complexity, but also supports a linear workflow layout in which actors are simply placed one after another in an intuitive order.

4.1 Inside Virtual Data Assembly Lines

Change-Resilience in Assembly Lines. In a physical assembly line, workers perform specialized tasks on products that pass by on the conveyor belt of a moving assembly line. Specifically, a worker only "picks" relevant products, objects, or parts thereof, letting all irrelevant parts flow through. Since each worker's *scope* is limited, a worker is unaware of the tasks of other workers and of the overall product being constructed. In particular, this has the advantage that a worker can be "reconfigured" to work on different parts of the object stream, and even moved up or down the assembly line, as long as certain inherent task dependencies are not violated.

By limiting work (via a scoping/configuration mechanism) to certain relevant parts of the object stream, and "passing the buck" on irrelevant parts, workers in an assembly line are *loosely coupled*, and the overall design is modular and resilient to changes. We employ and extend this processing paradigm to *data assembly lines* of streaming XML-like data.

VDAL data model. The data model for channels in VDAL are nested, labeled, ordered collections with metadata. This data model corresponds to XML. Labeled collections correspond to XML tags containing the collections' data, called *base data*. Domain-specific types, e.g., *PhylogeneticTree* or *CharacterMatrix*, can be represented as `CData`. Also the usual general-purpose types such as *Integer*, *Boolean* and *String* can be used in leaf nodes of the XML tree. Metadata corresponds to XML attributes, which can provide more information, e.g., the score of a sequence alignment, or can be used as simple annotations, e.g., the tag `faulty` could be attached to data or whole collections. Deploying XML as the data model naturally preserves data cohesion *and* allows efficient streaming of data when the XML tree is serialized and processed by actors in a SAX-like manner.

Moving data selection and shimming into configurations. Assume we want to place an actor A in a process network. If A has many input ports, then these must be wired to other actors (or shims) to describe the data routing (explicitly), leading to networks as shown in Fig. 6(a). If we instead design actor A to have one input port that expects data bundled in a custom record type α, then it is hard to place A into a network without explicit shims. If A's predecessor produces type τ objects and the successor step requires type τ' objects:

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

A conventional approach requires that $\tau \prec [\alpha]$ and $[\beta] \prec \tau'$, that is, the input stream consists of a list of α-compatible types $[\![\tau]\!] \subseteq [\![\,[\alpha]\,]\!]$ and the output stream $[\beta]$ has to be compatible with τ', i.e., $[\![\,[\beta]\,]\!] \subseteq [\![\tau']\!]$. However, these are very rigid constraints: In general A might not be able to accept τ instances (but require an adapter to filter the relevant part and/or to assemble the required α structure); similarly, β might not be of the desired result type τ'.

In contrast, in a virtual assembly line, each scientifically meaningful actor `A` is embedded in a framework of adapters as shown in Fig. 7. The data that flows into the actor is structured as an XML tree that maintains data associations. But instead of feeding the XML stream directly into the scientific actor `A`, configurable components around `A` select and package the data according to `A`'s input requirements. The inner functions of `A` are not understood by the workflow system, which simply invokes the component as a black-box. The behavior of the components σ, γ, ω, and M are determined by their configurations. With an appropriate formalism for these configurations, the workflow system itself can automatically analyze certain parts of the data flow in a workflow design—the components σ, γ, ω, and M can thus be viewed as white-box actors. However, the components σ, γ, ω and M need not be realized as explicit actors in the workflow specification. Instead their functionality can be provided automatically by the workflow system itself. Consequently, actors in VDALs are visually represented as normal actors, each with one input port and one output port, along with configurations for σ, γ, and ω; M does not have any configuration. Each of the components σ, γ, ω, and M is responsible for particular aspects of the data manipulation as detailed in the following subsection.

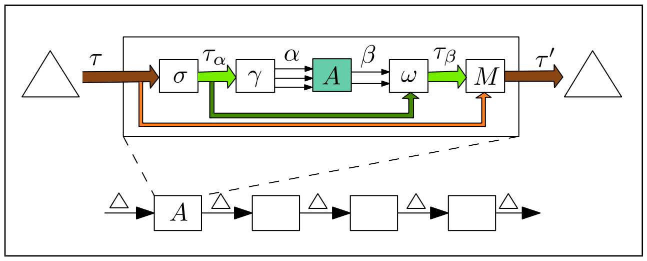


Figure 7: VDAL Actor Anatomy. In virtual data assembly lines, each black-box actor A is encapsulated between easily configurable, standard components σ, γ, ω that simplify data management and shimming. This allows flexible, localized data unpacking (via σ and γ) and re-packing (via ω), while requiring only one input and one output port through which XML-structured data is streamed.

4.2 VDAL Components and Configurations

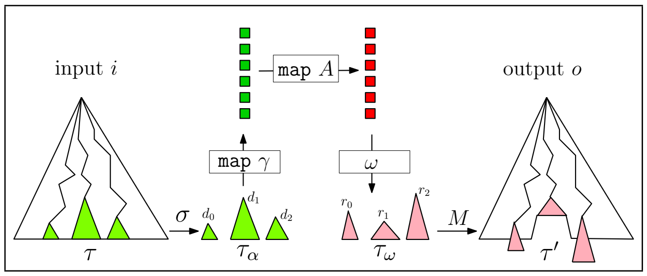


Figure 8: Dataflow inside VDAL Actor.

Figure 8 shows how data is manipulated as it flows through VDAL actors. The scope parameter σ determines the parts of the input data that are read and potentially modified by the actor. Then for each σ-selected subtree d_i, the input assembler γ stages all necessary input data to invoke `A`, possibly multiple times. The data to be staged is specified via configuration of γ and may be given as literal values or via path expressions that describe how to extract the data from the subtree d_i currently in scope. After the black-box `A` has been invoked on each of the staged sets of input data provided by γ, the write expression ω specifies how the scope d_i should be modified, usually by inserting the results of `A`'s invocations. The component M inserts the modified subtree d_i back into the output stream o. Usually, scopes are modified as they flow through the actor, and thus M happens implicitly. **Black-box actor A.**

The interface between a black-box actor `A` and the scientific workflow system is the list of the named input and output parameters to `A`. Each input and output parameter has a name and an associate type description. A type description consists of a name of a **BaseType**, i.e., `Integer` or `CharacterMatrix`, and an optional modifier "*" to indicate either that a list is required as input, or that a list is created as output.

Scope σ. The scope parameter σ selects relevant parts of the input stream τ. As in a physical assembly line, the actor does not read, or even modify anything in τ that has not been selected via σ. Formally, σ is a function from XML data to a list of relevant *read-scope matches* $[\tau_\alpha]$[3].

$$\sigma \ : \ \tau \rightarrow [\tau_\alpha]$$

The scope σ is specified via an XPath expression that uses *child* and *descendent* axes. Since we want to ensure that the selected scopes are non-overlapping, we use a *first-match* semantics for the descendent axis //. That means, a breath-first traversal that checks for scope matches will not traverse into an already found match. While we prohibit general side axis, checking the presence and/or values of attributes attached to nodes along the path is allowed. Prohibiting side axis allows the decision whether a subtree is a scope match or not to be made solely based on its ancestor nodes and their attributes. In a streaming implementation, for example, we can decide whether a certain subtree is a scope-match as soon as the root node of this tree is encountered (as all ancestor nodes and attributes have already been seen).

Input assembler γ. According to its configuration, the input assembler γ stages input for one or multiple invocations of the black-box actor A, using either data encountered inside each scope mach d_i or data provided as literals in the configuration.

If we represent one set of input values for A as a composite type α, then the input assembler γ is a function from τ_α to a list of α:

$$\gamma \ : \ \tau_\alpha \rightarrow [\alpha]$$

Each tuple in $[\alpha]$ is then provided to A, producing the output list $[\beta]$. As before, the black-box A is characterized as:

$$A \ : \ \alpha \rightarrow \beta$$

We suggest to use a query (or *binding expression*) for each input port of A. Each binding expression provides data either for a single invocation of A or for multiple invocations. Since black-box functions can expect a list of data per invocation, binding expressions usually select lists of lists. Formally, for each input port p_i the binding expression B_i represents a query that given the data in the read-scope $d_i \in \tau_\alpha$ produces a list of lists of base data suited for the port p_i.

$$B_i \ : \ \tau_\alpha \rightarrow [\,[T]\,], \qquad \text{with } T \in \textbf{BaseType}$$

The black-box A is then invoked once for each element of the Cartesian product

$$C := B_1(d_i) \times B_2(d_i) \times \cdots \times B_n(d_i), \qquad (\times)$$

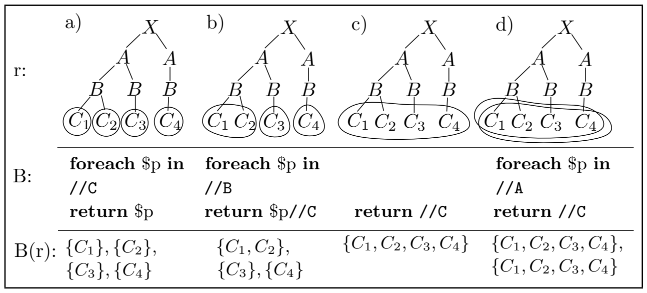
that is each element of C is of type α, which in turn is used to create an output tuple of type $\beta = A(\alpha)$.

We suggest to use the standard *foreach* loop with two XPath expressions to specify the binding expression queris:

foreach $\$p$ **in** XPath$_1$ **return** XPath$_2$

To be able to easily grab base-data, we imagine all BaseType-leaf nodes to be implicitely labeled with the type-name. Selecting these nodes via an XPath expression will select the actual value. Furthermore, in contrast to the usual XQuery semantics, we do not flatten

[3]We use $[x]$ to denote lists of type x.



r:
a)
b)
c)
d)
X
A
B
C_1 C_2 C_3 C_4
B:
foreach $p in //C return $p
foreach $p in //B return $p//C
return //C
foreach $p in //A return //C
B(r):
$\{C_1\}, \{C_2\}, \{C_3\}, \{C_4\}$
$\{C_1, C_2\}, \{C_3\}, \{C_4\}$
$\{C_1, C_2, C_3, C_4\}$
$\{C_1, C_2, C_3, C_4\}, \{C_1, C_2, C_3, C_4\}$

Figure 9: Example grouping via binding expression in γ.

the result sets to form one long output list, instead the result nodes from XPath$_2$ are grouped by the result of XPath$_1$, i.e., for each new node bound to p a new group is formed.

As example, consider the XML tree r as shown in Fig. 9. The C data that is available in the scope can be grouped in different ways: In Fig. 9(a), each C_i is put in a single group each of which will result in an invocation of the black-box actor. In Fig. 9(b), the results are grouped by B, i.e., all C's that are descendent of the same B node will be in a single group. Here, the black-box function would be called 3 times, once with each group as input. Only one group for one invocation is created in Fig. 9(c), whereas in Fig. 9(d) the same input data (all 4 C's) is presented twice to the black-box.

Literal values. Besides being able to select data from the read-scope, we suggest that, like in conventional dataflow models, literal values can also be put as parameters. We propose to extend standard conventions for integers (1, 34, -232), boolean values (true, false), strings ("foo", "bar"), or floating point numbers (0.2, -4.2e-7) with simple range constructs such as 1..10 (integers from 1 to 10) to facilitate simple parameter sweeps. Groups can easily be described using curly braces. Although it might be tempting to embed a small programing language here, we suggest to keep binding expressions rather simple. Using a Turing-complete language would significantly reduce workflow predictability and the effectiveness of static analysis for VDAL workflows.

Write expression ω and replacement M. The purpose of the write expression ω is to insert the results $[\beta]$ of the black-box function A into the scope τ_α, or to perform more drastic changes to the scope in order to produce τ_β. Here, an XML update language can be used. Formally:

$$\omega \ : \ [\beta], \tau_\alpha \rightarrow \tau_\beta$$

In the last step M, the modified scope τ_β replaces τ_α in the original stream τ to form the output τ' (see Fig. 8). In a streaming implementation, the replacement would be implicit as τ_α would typically be changed "in place" to form τ_β. Formally, M has the following signature:

$$M \ : \ [\tau_\beta], \tau \rightarrow \tau'$$

Example: VDAL Actor configurations. In Fig. 10, the configuration for an `CipresRAxML` actor is shown. The black box has five input parameters, and produces a list of phylogenetic trees. The actor's scope is //Nexus, such that input data is searched for only within subtrees labeled with `Nexus`. The service should be called for each method that is under a `Model` collection in the scope; the CharacterMatrix is also provided *somewhere* in the scope and selected via //CharacterMatrix. As rate categories two values, 25 and 100, should be used, and the initial rearrangement limit is set to 100. A cross-product of staged data upon which the actor is to be invoked is built from the multiple selected models and the specified seed values. The list of resulting trees is inserted within a new subtree labeled `Trees` inside the current scope match.

```
1  BlackBox: CipresRAxML
2    Input:   model of String
3             cha_matrix of CharacterMatrix
4             weight_vec of WeightVector*
5             rate_cats of Integer
6             init_rearr_limit of Integer
7    Output:  trees of PhyloTree*
8  σ: //Nexus
9  γ: model             ← foreach $p in //Model return $p/String
10    cha_matrix        ← //CharacterMatrix
11    weight_vec        ← //WeightVector
12    rate_cats         ← foreach $r in {25},{100} return $r
13    init_rearr_limit  ← 100
14 ω: INSERT AS LAST INTO . VALUE Trees[ $result/trees ]
```

Figure 10: Blackbox and VDAL actor configuration. Lines 1-7 describe the black-box embedded in the VDAL actor. In lines 8-14, Scope, Bindings, Iteration, and WriteExpression are used to specify how the black-box is fed with data from the incoming XML stream (σ, γ) and how the output of the black-box is placed back into the stream (ω).

5. DESIGN CHALLENGES REVISITED

Below we show how the VDAL modeling paradigm addresses the challenges presented in Sect. 3.

5.1 Parameter-rich Black Boxes

In Virtual Data Assembly Lines, parameters and inputs to black-box functions are not provided by individual ports nor is there a custom input structure necessary. Instead, VDAL extends the approach of regular parameters: Input can be specified either as literal values (just like with regular parameters) or as special path expressions that grab the data from the actor's input stream. VDAL actors thus exhibit only one input and one output port, through which the XML-stream of data flows, reducing necessary wiring to a minimum.

Reduction of workflow graph complexity. Of course, the input for a black-box still needs to be specified somehow; our approach *moves* moves the scientifically essential portion of the complexity from the graphical wiring into the configurations. Moreover, it completely removes from the model *non-essential complexity*, i.e. all explicit references from one actor to another. In a conventional workflow, a wire directly connecting one actor to another expressly indicates that the output of the first actor is to be consumed and processed by the second actor. In a VDAL workflow, in contrast, a wire directly connecting two actors by no means implies that the downstream actor uses any information transmitted by the actor over that wire. The order of actors in a VDAL workflow merely indicates the order in which actors will have access to the data stream. The wires between actors serve only as the channel over which the entire data stream passes between actors and do not indicate direct interactions between connected actors.

A further way in which configurations are superior to explicit routing in selecting necessary input data is that configurations are declarative descriptions. They specify what data to use from where in the stream in contrast to the operational descriptions of wires and record-management shims. To select all character matrixes inside the current read scope, for example, an XPath expression "//CharacterMatrix" can be used. Not only is this more concise than one or more shim-actors for selecting data from a record-structures, it also makes the actor oblivious to certain changes in the input stream, and consequently makes the behavior and configuration of the actor more resilient to future changes in the effective schema of the incoming data. For example, additional data items or deeper nesting can be accommodated without changing the configuration of the actor.

Example. With the flexible adapters σ and γ available inside the actor, the `CipresRAxMLService` can just be inserted into the data assembly line (shown in Fig. 11). Via a configuration as shown in Fig. 10, the input data for the black box is selected from the incoming XML stream.

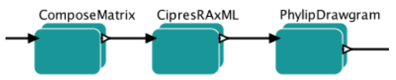


Figure 11: Linear workflows. In Data Assembly Lines, even parameter-rich services are connected with only one input and one output channel. Individual input data are specified as literal values or as an XPath expression, which extracts data from the incoming stream. A sample configuration for the `CipresRAxML` actor is shown in Fig. 10.

5.2 Maintaining Data Cohesion

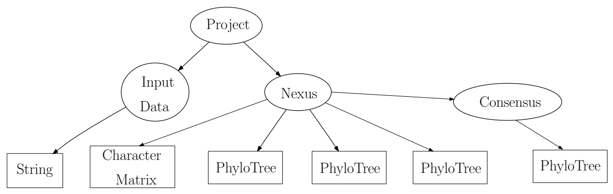
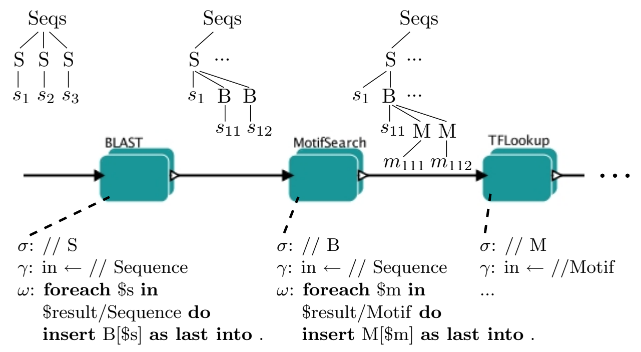


Figure 12: Hierarchical data used in phylogenetic workflow. Data-nodes (domain-specific and general purpose data) are shown as rectangular boxes, collection labels as ovals.

The XML data model of VDAL can directly be used to maintain relationships between data items. During workflow execution, access to specific parts of the data is provided by the query capabilities of σ and γ. Data associations that are maintained in custom records, or domain-specific file formats, such as FASTA or Nexus files, can be modeled via the XML tree structure. Figure 12 shows how the data processed by a phylogenetic workflow could be organized. Starting from the left side of the figure, the String data contains a URL that points to a Nexus file expected to contain a character matrix. The workflow fetches the file, converts it into a CharacterMatrix domain-type, and stores this data item in a Nexus collection. Via the `CipresRAxML` actor several PhyloTrees are inferred and placed in the same Nexus collection; in a last step, a consensus tree is computed and stored under a Consensus collection. The `PhylipDrawgram` actor, which displays phylogenetic trees, could then easily be configured to either draw all the trees in the workflow (σ = //PhyloTree), to draw only the trees inferred via `CipresRAxML` (σ = //Nexus/PhyloTree), or to draw only the consensus tree (σ = //Consensus/PhyloTree).

Data cohesion for nested lists. Let us reconsider the example presented in Sect. 3.2: The services `BLAST`, `MotifSearch`, `TFLookup`, and `FunctionLookup` each produce a list of output items whenever one input item is received. To maintain the associations between items produced by actors and the inputs from which they were derived, individual data tokens were wrapped into array tokens. This leads to type-mismatches on the input ports of the actors if they are simply chained together. We thus had to introduce `Loop` actors that essentially perform a `map` over these lists. In data assembly lines, associations can be maintained using nested XML structures, from which the data is selected via configuration parameters. Since these parameters can be specified using the descendent axis, the actor is decoupled from the actual nesting depth of the organizational structure.

Figure 13 shows the workflow from Sect. 3.2 modeled as a data assembly line. As input data, we place each sequence s_i inside an



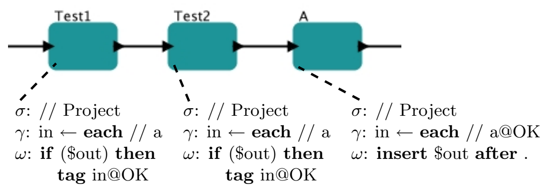
```
σ: // S
γ: in ← // Sequence
ω: foreach $s in
     $result/Sequence do
     insert B[$s] as last into .

σ: // B
γ: in ← // Sequence
ω: foreach $m in
     $result/Motif do
     insert M[$m] as last into .

σ: // M
γ: in ← //Motif
...
```

Figure 13: Maintaining nesting structure and automatic `map`

S-collection. The `BLAST` actor's scope σ selects each of the collections in turn. Input assembler γ selects the one sequence inside the scope and invokes the black-box function to create the list of sequences as output. Via the write expression ω, we then insert a B-collection for each output sequence into the S-collection. The resulting XML structure is shown over the channel leading from `BLAST` to `MotifSearch`. The second actor then analogously creates M-subcollections for each motif that was associated with the "BLAST-ed" sequences. Succeeding actors follow the same design idea. Note that this approach not only removes explicit loop-actors, but also enables each actor to select the correct input data independently of how deeply nested within the input data stream it is located. It is thus easy to add additional nestings to the input schema. A top-level "Projects" collection could, for example, be introduced to hold multiple "Seqs" collections, each containing, say, sequences from different groups organisms. During the workflow execution, this organizational structure would be kept intact. Furthermore, we could add additional information into the XML tree without disturbing the actors that do not need to access it. Since γ *selects* the data that is relevant from the input, additional data is ignored without the need of further routing actors.

5.3 Conditional Execution

In data assembly lines, conditional execution and the necessary data routing is localized to the adapters σ, γ and ω. Consider the use-case from 3.3, in which an actor `A` should be executed on some data d only if at least one of two tests (perfomed by actors `Test1` and `Test2`) were successful. Instead of using different routes for the data, we use the actors `Test1` and `Test2` to *tag* the data items with the result of the test. Then, we exploit the querying capabilities of σ and γ to only select these data items for which one of the test results are positive. The other data is simply ignored and passed down the data assembly line. In a sense, the routing around the actor `A` is kept local (inside the configuration shell of `A`) while the information originating from the test actors `Test1` and `Test2` is attached to the data.



```
σ: // Project
γ: in ← each // a
ω: if ($out) then
     tag in@OK

σ: // Project
γ: in ← each // a
ω: if ($out) then
     tag in@OK

σ: // Project
γ: in ← each // a@OK
ω: insert $out after .
```

Figure 14: Localizing if-then-else routing via XML attributes

5.4 Iterations over Cross Products

Via parameters for the scope and input assembler, the workflow developer declares which data is used as input for the black boxes. Using the **foreach** construct, cross-products can easily be declared without the need of explicit routing and token repetition. Input data for these multiple black box invocations can be specified as parameter via literal values in the binding expressions, or inside the input data stream of the actor.

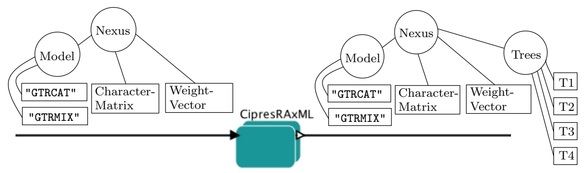


Figure 15: Cross-products in VDAL. Performing cross-products is a built-in feature of the VDAL configuration layer. Thus, no shims or complex routing is necessary. Here, the black-box component is invoked four times according to the configuration given in Fig. 10: two different models and two values for the rate_cats parameter. Four output trees are placed under the newly created `Trees`-collection.

Figure 15 shows how the `CipresRAxML` actor with a configuration as in Fig. 10 transforms an incoming data stream: The binding expressions will select the CharacterMatrix and WeightVector inside the Nexus collection. For the model parameter, all Strings under //Model (here `"GTRCAT"` and `"GTRMIX"`) are selected inside the **foreach** construct. For the rate_cats input, multiple parameters are provided as literal values. The CipresRAxML service is invoked four times, as there are two values for the model (`"GTRCAT"` and `"GTRMIX"`) and two values for the rate_cats parameter (25 and 100). The resulting trees, are placed inside a new collection labeled with Trees, according to the write expression ω. Since input creation and iterative invocation of the black-box is part of the workflow infrastructure, no explicit loops, or repeater shims have to be placed around the actor.

6. DISCUSSION AND RELATED WORK

In the scientific workflow community, workflow design issues have received comparatively little attention in the past, but their importance is now more fully emerging [4, 10, 15, 13, 18, 19, 1, 35]. In this paper, we have presented VDAL, a scientific workflow modeling and design paradigm for data assembly lines that aims at minimizing the "shimantic web syndrome" [16], i.e., the proliferation of unnecessarily complex workflow designs that involve large numbers of shim actors and "messy wiring", thus obfuscating the scientific protocol that the workflow should capture in the first place. VDAL borrows ideas from, among others, flow-based programming [26], XML querying and stream processing [38] (e.g., scope σ), functional programming (e.g., `map` γ), and, most importantly, Kepler/COMAD [23, 24]. At the heart lies the idea of a virtual data assembly line, where nested data collections are streamed through a largely linear chain of VDAL actors, each of which has a built-in, easily configurable data access and management layer for selecting a substream of relevant input elements (σ), from which concrete data inputs can be subselected and reorganized (γ), before being fed to the innermost scientific black-box function (A), placing A's results at suitable positions in the output stream (ω). This architecture (i) eliminates many "data massaging shims" as their functionality is instead part of the actor configuration, given by the standard operations (σ, γ, ω), and (ii) minimizes *non-local wiring*[4]: e.g., in Figure 4 the actors `BooleanSwitch` and `DetermMerge` are directly connected via one channel and non-locally wired through

[4]A connected pair of actors $A \rightarrow B$ has non-local wiring, if there is an alternate, indirect path $A \rightarrow \cdots \rightarrow B$ between them.

a subworkflow A. In contrast, VDAL workflow designer can understand a workflow locally, by inspecting its actor configuration; global effects, on the other hand, can be inferred using static analysis if necessary [38].

A limitation of a strict assembly-line is that data products are flowing strictly downstream. To iteratively execute a certain number of actors (in a while-loop-like fashion), the paradigm needs to be extended by looping constructs that route data back to earlier steps in the workflow. Although our Kepler/COMAD implementation already provides (some) looping support, a clear theoretical investigation should be performed in future work.

Acknowledgements. This work was supported in part by NSF awards IIS-0612326, OCI-0722079, DBI-0619060, DE-FC02-07ER25811, and IIS-0630033.

7. REFERENCES

[1] D. Abramson, B. Bethwaite, C. Enticott, S. Garic, and T. Peachey. Parameter space exploration using scientific workflows. In *Intl. Conf. on Computational Science*, LNCS 5544, pages 104–113, 2009.

[2] R. Armstrong, D. Gannon, A. Geist, K. Keahey, S. Kohn, L. McInnes, S. Parker, and B. Smolinski. Toward a common component architecture for high-performance scientific computing. In *In Intl. Sympos. on High Performance Distr. Computation*, 1999.

[3] S. Bowers and B. Ludäscher. An ontology driven framework for data transformation in scientific workflows. In *Intl. Workshop on Data Integration in the Life Sciences (DILS)*, LNCS 2994, 2004.

[4] S. Bowers and B. Ludäscher. Actor-oriented design of scientific workflows. In *Intl. Conf. on Conceptual Modeling (ER)*, LNCS 3716, 2005.

[5] 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)*, Atlanta, GA, April 2006.

[6] S. Bowers, T. McPhillips, S. Riddle, M. Anand, and B. Ludäscher. Kepler/pPOD: Scientific workflow and provenance support for assembling the tree of life. In *Intl. Provenance and Annotation Workshop (IPAW)*, 2008.

[7] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java (volume 1: Introduction to ptolemy ii). Technical Report No. UCB/EECS-2008-28, April 2008.

[8] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. *Workflow Management in Condor*, pages 357–375. In Taylor et al. [33], 2007.

[9] S. B. Davidson, S. C. Boulakia, A. Eyal, B. Ludäscher, T. M. McPhillips, S. Bowers, M. K. Anand, and J. Freire. Provenance in scientific workflow systems. *IEEE Data Engineering Bulletin*, 30(4):44–50, 2007.

[10] E. Deelman. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[11] E. Deelman, D. Gannon, M. Shields, and I. Taylor. Workflows and e-Science: An overview of workflow system features and capabilities. *Future Gen. Computer Systems*, 25(5):528–540, 2009.

[12] J. Felsenstein. PHYLIP (phylogeny inference package) version 3.6. *Distributed by the author. Department of Genome Sciences, University of Washington, Seattle*, 2004.

[13] J. Freire, C. Silva, S. Callahan, E. Santos, C. Scheidegger, and H. Vo. Managing rapidly-evolving scientific workflows. In *Intl. Provenance and Annotation Workshop (IPAW)*, LNCS 4145, pages 10–18, 2006.

[14] 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 *National Conference on Artificial Intelligence*, pages 1767–1774, 2007.

[15] J. Hidders, N. Kwasnikowska, J. Sroka, J. Tyszkiewicz, and J. V. den Bussche. Petri net + nested relational calculus = dataflow. In *OTM Conferences*, pages 220–237, 2005.

[16] D. Hull, R. Stevens, P. Lord, C. Wroe, and C. Goble. Treating "shimantic web" syndrome with ontologies. In *Advanced Knowledge Technologies Workshop on Semantic Web Services*, 2004.

[17] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Proc. of the IFIP Congress 74*, pages 471–475. North-Holland, 1974.

[18] D. Koop, C. Scheidegger, S. Callahan, J. Freire, and C. Silva. Viscomplete: Automating suggestions for visualization pipelines. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1691–1698, 2008.

[19] 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.

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

[21] B. Ludäscher, I. Altintas, S. Bowers, J. Cummings, T. Critchlow, E. Deelman, D. D. Roure, J. Freire, C. Goble, M. Jones, S. Klasky, T. McPhillips, N. Podhorszki, C. Silva, I. Taylor, and M. Vouk. Scientific process automation and workflow management. In A. Shoshani and D. Rotem, editors, *Scientific Data Management: Challenges, Existing Technology, and Deployment*, Computational Science Series, chapter 13. Chapman & Hall/CRC, 2009.

[22] B. Ludäscher, S. Bowers, and T. McPhillips. Scientific workflows. In *Encyclopedia of Database Systems*. Springer, 2009.

[23] T. McPhillips and S. Bowers. An Approach for Pipelining Nested Collections in Scientific Workflows. *SIGMOD Record*, 34(3):12–17, 2005.

[24] T. McPhillips, S. Bowers, and B. Ludäscher. Collection-oriented scientific workflows for integrating and analyzing biological data. In *3rd Intl. Workshop on Data Integration in the Life Sciences*, 2006.

[25] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific workflow design for mere mortals. *Future Generation Computer Systems*, 25(5):541–551, 2009.

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

[27] 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*, 18(10):1067–1100, August 2006.

[28] N. Podhorszki, B. Ludäscher, and S. Klasky. Workflow Automation for Processing Plasma Fusion Simulation Data. In *2nd Workshop on Workflows in Support of Large-Scale Science (WORKS)*, June 2007.

[29] D. D. Roure, C. Goble, and R. Stevens. The design and realisation of the myexperiment virtual research environment for social sharing of workflows. *Future Generation Computer Systems*, 25:561–567, 2009.

[30] A. Stamatakis, M. Ott, and T. Ludwig. RAxML-OMP: An Efficient Program for Phylogenetic Inference on SMPs. *Lecture Notes in Computer Science*, 3606:288–302, 2005.

[31] K. Stevens, D. Cutler, M. Zwick, P. de Jong, K. Huang, M. Koriabine, B. Ludäscher, C. Marston, S. Lee, D. Okou, K. Osoegawa, J. Warrington, D. Begun, and C. Langley. DPGP cyberinfrastructure and open source toolkit for chip based resequencing. In *Advances in Genome Biology and Technology (AGBT)*, 2007.

[32] Taverna. http://taverna.sourceforge.net.

[33] I. J. Taylor, E. Deelman, D. B. Gannon, and M. Shields, editors. *Workflows for e-Science: Scientific Workflows for Grids*. Springer, 2007.

[34] Triana project. http://www.trianacode.org.

[35] K. van Hee, J. Hidders, G.-J. Houben, J. Paredaens, and P. Thiran. On the relationship between workflow models and document types. *Information Systems*, 34(1):178–208, March 2009.

[36] D. Zinn, S. Bowers, S. Köhler, and B. Ludäscher. Parallelizing XML processing pipelines via MapReduce. *Journal of Computer and System Sciences*, 2009. Special issue on Scientific Workflow. Accepted for publication.

[37] D. Zinn, S. Bowers, and B. Ludäscher. XML-Based Computation for Scientific Workflows. In *Intl. Conf. on Data Engineering (ICDE)*, 2010. to appear.

[38] D. Zinn, S. Bowers, T. M. McPhillips, and B. Ludäscher. X-CSR: Dataflow Optimization for Distributed XML Process Pipelines. In *Intl. Conf. on Data Engineering (ICDE)*, pages 577–580, 2009. Also see Technical Report CSE-2008-15, UC Davis.