

XML-Based Computation for Scientific Workflows

Daniel Zinn^{*}, Shawn Bowers[‡], Bertram Ludäscher^{*†}

^{*}Department of Computer Science

[†]UC Davis Genome Center

University of California, Davis

[‡]Center to Dept. of Computer Science

Gonzaga University

{dzinn, sbowers, ludaesch}@ucdavis.edu

Abstract—Scientific workflows are increasingly used for rapid integration of existing algorithms to form larger and more complex programs. Such workflows promise to provide more abstract, yet executable views of data analyses through graphs of components representing computational tasks and edges representing the data-flow between tasks. However, designing workflows using purely data-flow-oriented models of computation introduces a number of challenges, including the need to introduce low-level components to mediate and transform data (so-called *shims*) and a large number of additional “wires” for routing data to components within a workflow. To address these problems, we present a modeling paradigm *Virtual Assembly Lines (VAL)* that can not only eliminate most shims and reduce the wiring complexity, but also provide significant help to scientists during workflow construction and maintenance. The core idea of VAL is to organize data on workflow channels into XML-like tree-structures (i.e., similar to file folders), and to employ an additional configuration shell around scientific components. This has the effect that much of the wiring complexity is moved into the configuration layer. We argue that a carefully crafted domain-specific language together with a powerful type system is superior to current approaches. In particular, we present a concrete instance Δ -XML of the general VAL framework, which leverages existing work on XML update languages to provide additional modeling benefits via static analysis.

I. INTRODUCTION

With computation and simulation gaining importance in many sciences, scientists are faced with the problem of integrating different software components. An increasing number of algorithms are available as down-loadable packages or web-services, and scientists often need to mix and match many of these components to perform complex analyses. In phylogenetic research, for example, different DNA sequences are retrieved from data bases, then aligned to a reference sequence before a set of phylogenetic trees is computed. These trees are then combined to a consensus tree which is then displayed to the scientist for manual inspection. For each of these steps (alignment, tree inference, consensus generation) specialized algorithms exist. Integrating these software components into a running system poses challenges. Since components are often written in different languages they cannot easily be used as libraries, and are therefore used as stand-alone programs. They often operate on text files with specific input formats and thus their composition is often done by hand (calling one program after another) or via simple shell or perl scripts. Some components (e.g., [12]) are also available as web-interfaces on

the Internet, where scientists manually insert parameters and upload input files before having the results e-mailed to them. A significant amount of components is also available as standard web-services. However, using these is not trivial for domain scientists that are not confident in programming. Many domain scientists are much more versatile within their field of research than with software-engineering techniques, and therefore they often choose to manually “integrate” these components, and consequently spend time with mundane tasks.

During the last years, scientific workflow systems have been proposed as a general approach for helping scientists with these integration task. So does Taverna [36] or Kepler [24], for example, allow to build workflows that combine locally available programs, which might be written in different languages, with programs that are accessed via web-service techniques. Once an existing algorithm has been wrapped as an *actor* in these systems, it can inter-operate with other actors without manual intervention—the details about its invocation is hidden from the domain scientist, which now can focus on composing these components. In the case of web-services, the wrapping actors can be created automatically from the web-service description provided in WSDL.

Common systems employ a dataflow-oriented paradigm: computational steps are represented as nodes connected via channels in a dataflow graph. The scientist then builds more complex analyses by placing nodes on a canvas and connects them with each other using a scientific workflow tool.

Despite this abstraction, it is still hard to construct complex workflows in current scientific workflow systems. A major problem is that larger workflows tend to have many channels and thus very complex workflow graphs. Complex wiring is often coupled with additional actors that are necessary for data manipulation, complex control flow or error handling. These non-scientific actors (or *shims*) further increase the complexity of the workflow itself. Large workflow graphs (many channels, mixed normal and shim actors) are hard to construct, extend, maintain and even to comprehend.

Contributions. In this paper, our contributions are as follows: (1) We describe common use-cases for scientific-workflow design and evolution. Furthermore, (2) we present an abstract modeling paradigm, called *Virtual Assembly Lines (VAL)* that addresses these use-cases. Specifically, VAL does not only

eliminate shims including most control-flow-actors and thus reduces the wiring complexity, but it also provides significant help to the scientist during workflow construction and maintenance. Our further contributions are to propose a specific instance Δ -XML of the VAL model for which we (3) show how it can be compiled to FLUX programs, and (4) how the FLUX type system can be used to provide additional features for scientific workflow designers, such as actor-dependency analysis.

The rest of the paper is structured as follows: In Sect. II we present current dataflow approaches and modeling challenges. In Sect. III, we explain the general framework of virtual assembly lines and explain how it addresses these challenges. Then, in Sect. IV, we present the Δ -XML instance of VAL, its compilation to FLUX and how the FLUX type system has to be adapted and used to provide additional modeling features. We conclude in Sect. V with a discussion of our and related work.

II. DATAFLOW-ORIENTED APPROACHES

Current scientific workflow systems, such as myGrid/Taverna [36], VisTrails [17], Chimera [16] and Kepler [6], are based on the paradigm of dataflow-oriented programming: Single tasks are represented as modules, or *actors*, which have designated input and output interfaces, called *ports*. Via a graphical user interface, complex analyses, *i.e.*, *workflows*, are assembled by connecting output ports to input ports. These connections are called *channels*. We are often referring to this graph (in which the nodes are actors and edges are channels) as the workflow graph. When a workflow is executed, data is flowing in between actors over the *channels*. The execution of a workflow is determined by its graph, the interior of the actors and the so-called *model of computation*, which defines further details about dataflow, actor invocations and workflow scheduling.

A very general model of computation are Kahn Process Networks [22]. Here, the channels are implemented as FIFO queues. Actors can be implemented as arbitrary programs that read from and write to their ports. It has been shown that the workflow result is deterministic if actors themselves are deterministic and if all reads block in case no data is available on the input port¹.

Other commonly used models of computations refine or restrict Process Networks in certain ways. In Synchronous Dataflow Networks, for example, actors follow a strict read, execute, write model in which the patterns for reading and writing are pre-defined. This helps to guarantee bounded buffer sizes on the channels and also allows to construct a serial schedule.

In general, workflows are similar to stream-processing system as they follow a push-model, but there are severe differences. Actors perform work when they receive input data and not when output data is requested (as it is done in

standard query processing). In contrast to stream-processing, however, actors are black-boxes for the workflow engine. Often times, these black-boxes wrap very sophisticated scientific methods, which might be data or computational expensive or both. Furthermore, invocation of the black-boxes is not as homogeneous as in regular stream processing systems: The functionality might be implemented behind a web-service or even as standalone programs on dedicated cluster computers. Furthermore, black-box functions often require sets of scientific data as input and produce objects of scientific data types. A RAXML [35] actor, which infers phylogenetic trees from input sequences, for example requires a set of aligned DNA or protein sequences from several species. Its output are different possible trees with confidence levels. While many of the research ideas from stream processing can be transferred to scientific workflows, it is important to keep these differences in mind.

Because scientific workflows are used to perform and support research, their creation is of a more exploratory nature than stream-processing systems. It is therefore essential to not only provide means for their fast execution, but also to support the scientist while creating and modifying the workflow. In the following, we will survey challenges that occur during the design of scientific workflows in dataflow oriented models of computation, such as Process Networks, or Synchronous Dataflow Networks.

A. Challenges in Workflow Design

CHALLENGE 1: Parameter-rich black-box functions. Many scientific components require a variety of different input data to work properly. Besides the actual input data, many parameters need to be specified. The black-box RAXML [35] for generating large phylogenetic trees, for example, not only takes the gene or protein sequences as input, but also additional parameters including initial seeds for the search, number of iterations or general model assumptions. DNAML (DNA Maximum Likelihood) from the Phylogeny inference package [15] takes 10 additional parameters besides the list of species and DNA sequences.

Consequently, actors wrapping these components have many input ports, each of which connected to a single channel. This quickly leads to many wires when several of these components are used. Sometimes, components expect all input data bundled together in a single large record data structure and they thus have only one input port. A prominent example are document-style web-services that expect one large SOAP message and output another one. Although these components do not exhibit any input or output ports, the problem is not solved since assembling these complex records is usually performed by shim-actors. Taverna, for example, provides the feature to automatically construct these shims based on the WSDL of the web service. Resulting workflows then often have shims to assemble and disassemble messages sent to scientific components. Here, the complex wiring happens between shim actors.

¹Actors that behave like this are monotonous functions on their input streams. The workflow output is then well-defined via a fixed-point computation, which exactly corresponds to running the workflow. [22]

CHALLENGE 2: Hierarchically organized scientific data. Scientific data products are often related to each other and structured in hierarchies. Maintaining and leveraging these associations between data is challenging and can lead to shim actors for packaging. The BLAST [4] service to search for similar gene or protein sequences, for example, defines an XML DTD to describe its hierarchical output data. Being able to access single fields or multiple related fields is essential for composing workflows from individual actors. Furthermore, workflows seldom perform only one single call to complex services. It is, for instance, quite common to select different gene or protein sequences from a set of species to identify regions of the inferred phylogenetic trees that are stable across the selected sequences. Here, it is essential to maintain associations between sequences, inferred trees, and the set of sequences they originate from.

If native support for hierarchical data is missing, hierarchical data is often represented as array or record datatypes, or even in plain strings or files as required by the black-box services. Assembling record and array structures and accessing fields of them require additional shim actors in the workflow. The shims that are necessary to dissect strings or files are often even targeted to the specific format and not only clutter the workflow graph but also require additional development efforts. Taverna natively supports arbitrarily nested lists of data items [32]. While this removes array-related shims in the workflow, labeled hierarchical data still needs to be assembled in shim actors.

CHALLENGE 3: Flexible grouping of data. Because many important scientific components do not work on only single data items but on collections of data (a set of gene-sequences is required for alignment), scientific data needs to be grouped when it is provided to the actor. In hierarchically organized data there are often multiple groupings possible (one group of sequences for all species, or separate groups for distinct subsets of species, *e.g.*, one for all mammals and one for all bacteria under consideration).

In plain dataflow approaches, these groups would be constructed as array structures or lists using shim actors. If the data has to be extracted from record structures, then the necessary filtering, once again, makes the workflow overly complex and tedious to develop.

CHALLENGE 4: Non-trivial control-flow. Looping and conditional executions are essential in many scientific analyses. Scientific components are often invoked multiple times during the execution of a single workflow, either on the same data with different parameters (for performing parameter studies or parameter searches) or on different data items or sets. Thus, loops require iterated execution of black-boxes with changing parameters, data sets or both.

In current models, control-flow constructs (loops and alternatives) are encoded into the dataflow network. A for-loop for example would use a “for-loop” actor that would emit integer tokens, which are then together with the necessary data routed to other actors that perform analysis tasks. Also,

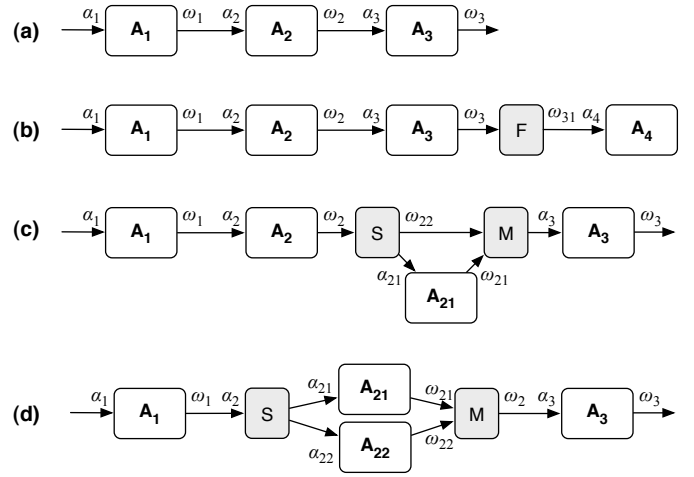


Fig. 1. Small changes to a simple design (a), require “glue components” (*adapters* or *shims*), obfuscating the conceptual design with low-level details (b–d).

if-then-else constructs are mapped into the dataflow graph and manifest into a control flow actor and two distinct routes. These control-flow actors together with their necessary wiring lead to complex designs for moderately-sized workflows [33].

CHALLENGE 5: Evolutionary design. To save developers’ time, it is important that scientific workflows can be created, maintained and modified easily. While this is a general goal for any sort of software system, it requires increased attention in the area of scientific workflows. Here, complex analysis are frequently modified, for example to try out different algorithms to perform certain steps in an analysis, or to obtain input data from different sources, or to combine existing algorithms in novel ways. Furthermore, modularity gains on importance because sharing workflows or parts thereof for re-use can not only save collaborating scientists development time, but also increases collaboration.

Current dataflow-oriented methods, however, do not facilitate easy modifications mainly due to the rigid typing that is deployed on the workflow channels. Consider the simple process network in Figure 1(a), where each actor $A_i : \alpha_i \rightarrow \omega_i$ has an input type α_i and an output type ω_i . In conventional approaches, the connection between actor A_i and A_{i+1} must satisfy a subtyping constraint $\omega_i \prec \alpha_{i+1}$. This rigid typing approach can result in “brittle” designs, *i.e.*, pipelines that are difficult to modify and evolve. For example, adding new steps or replacing existing ones usually requires the addition of shims to ensure type safety and correct execution. The following simple example illustrates some of the problems in the traditional typing approach.

Example (Pipeline Evolution). In Figure 1(b) we wish to add a new actor $A_4 : \alpha_4 \rightarrow \omega_4$ to the end of the pipeline of Figure 1(a). 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 pipeline to filter the output of A_3 thus obtaining the parts needed by A_4 .

In Figure 1(c), we wish to add a new processing step, im-

plemented by actor A_{21} between two existing actors. Similar to the previous case, A_{21} works only on specific parts of the output of A_2 , and further, only produces a portion of the desired subsequent input type α_3 . Here, we must add two new actors to satisfy the type constraints: an actor S is added to split the output of A_2 into both the parts required by A_{21} and the remaining “non-relevant” parts; and an actor M is added to merge the output of A_{21} with the remaining output of A_2 , which is then used as input by A_3 .

In Figure 1(d), we wish to replace the existing 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 actors to appropriately split and merge the stream.

B. Drawbacks of Excessive Wiring and Shims

Using many shims with their necessary wiring makes constructing scientific workflows not only tedious, but resulting workflows are also hard to understand, extend or maintain in general. Having many shims and control-flow actors in a scientific workflows distracts from the main scientific software components encapsulated in normal actors and makes it thus hard to identify the workflows’ main computations. Furthermore, the wiring often leads to non-planar graphs making it hard to spot where the data flows. In addition, channels that only carry control-flow tokens distract from channels with scientifically meaningful data. Taverna addresses this issue by allowing scientists to declare actors as “boring”, which hides those actors from the canvas. Unfortunately, the shims are only visually hidden. New channels or changes to the workflow design still require connecting hidden shims with other actors or shims. Excessive wiring and shims not only obfuscate workflows, but make it also hard to modify them. Placing new scientific actors requires a complete understanding of the current wiring, which often is complex. Also, removing existing actors will most certainly break the current design as formerly connected channels are invalidated.

In the next section, we will present an approach that does not require any shims to address the challenges described above. We will also show how this new approach can actively help workflow designers with adding and removing of actors to/from existing workflows.

III. VIRTUAL ASSEMBLY LINES (VALS)

We combine ideas from process networks and stream processing with XML-structured data and XML update languages. Specifically, our approach, *Virtual Assembly Lines (VALs)* is characterized by four core ideas: Linear Process Networks, XML-structured data on the channels, powerful actor configurations, and static analysis of workflows (see Fig. 2). (1) *Linear Workflows*. A virtual 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 on channels is structured in labeled trees possibly with additional attributes much like XML data. The data is streamed in a SAX-like manner on the channels,

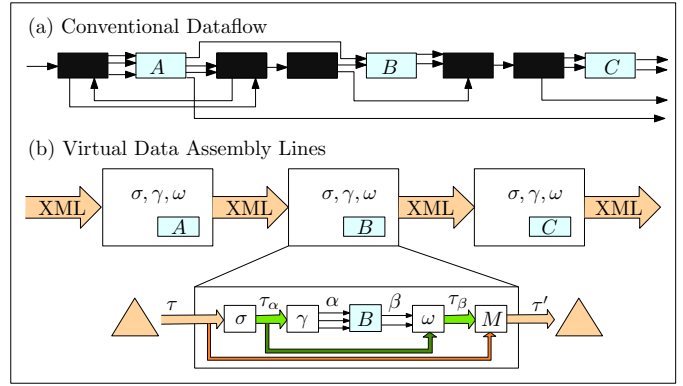


Fig. 2. In Virtual Assembly Lines, data messaging is moved to a configuration layer, denoted by α, ϕ, ω . This reduces wiring complexity and facilitates linear workflows with re-usable components.

although different execution strategies are thinkable². This is in contrast to common approaches where data on channels is of simple or custom record data types. (3) *Configuration shell*. Scientific components are wrapped in a white-box-layer, in which scientists can configure where and 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 deploy a domain-specific language to eliminate most data messaging tasks that are performed in record-assembler and disassembler shims nowadays. Moving the data *messaging* into the configuration layer not only reduces the wiring complexity, but also allows the linear workflow structure in which actors are simply placed one-after-another. (4) *Static type system for modeling support*. Virtual assembly lines can be statically analyzed to provide valuable information to the scientists during workflow creation and maintenance. It is for example possible to estimate the effects of removing or adding actors to the overall workflow.

A. Linear Workflows with Structure-Rich Channels

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. For example, consider a car assembly line with workers A_m (attaching mirrors) and A_b (attaching bumpers). Clearly, we can commute A_m and A_b , *i.e.*, swap their locations along the assembly line, without affecting the final product. Similarly, workers can be removed or new workers inserted without

²And are for example explored in [38] or [37].

breaking the overall pipeline, provided there are no scoping conflicts. For example, we cannot remove the worker putting on axles and expect a downstream worker to attach wheels. For the same reason, those two workers cannot be swapped.

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 *virtual assembly lines* of streaming XML data.

Moving data massaging into configurations. Assume we want to place an actor A in a process network. In case A has many different input ports, they need to be wired up to other actors (or shims) to describe the data routing (explicitly), leading to networks as shown in Fig. 2(a). If we design actor A to have one input port that receives the data bundled to a custom 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:

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

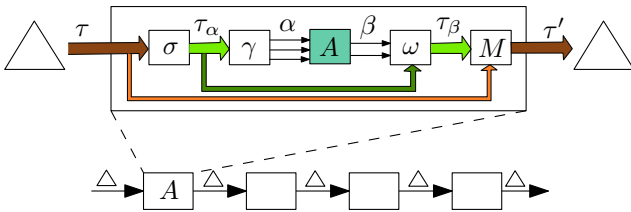
A conventional approach requires that

$$\tau \prec [\alpha] \quad \text{and} \quad [\beta] \prec \tau', \quad (\prec)$$

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.*, $[[[\beta]]] \subseteq [[\tau']]$.

As illustrated in the example above, 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 subsequent type τ' .

In contrast, in a virtual assembly line, the actor inside can be visualized as follows:



We move data massaging into a configuration-shell around the black-box actor A . Via parameters for read-scope σ , iteration scope ϕ , and write-scope ω , we define how α -typed inputs for A are constructed from the input stream τ and how the results β of applying A to α are placed back into τ to form the output stream τ' .

B. VAL-Configurations

Read scope σ . A read-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

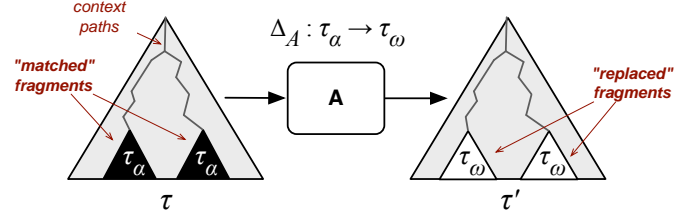


Fig. 3. The scope of actor (stream processor) A is given by a configuration Δ_A with read-scope σ . By σ selected parts τ_α are replaced by new subtrees τ_β , which are created by combining the results from black-box A with τ_α .

not been selected via σ . Formally, σ is a function from XML data to a list of relevant *read-scope matches* $[\tau_\alpha]$.

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

In Δ -XML, we will use some fragment of XPath to describe σ . As shown in Fig. 3, these subtrees of τ are then modified or completely replaced by the actor.

Iteration scope ϕ . The iteration scope parameter defines how to create input data for A from an individual read-scope match τ_α . From one τ_α , the iteration scope ϕ can create multiple input sets for several invocations of A . This is typically done by selecting the input data from τ_α or by providing parameter values as literals. As an example, building the cross-product over multiple input values, as proposed later in Δ -XML, is an elegant way to define parameter sweeps.

Like above, we represent one set of input values for A as type α . Thus, iteration scope ϕ is a function from τ_α to a list of α :

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

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

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

Write scope ω and replacement M . The write-scope configuration ω determines how the obtained list of output values $[\beta]$ is combined with the read scope τ_α to form the modified scope τ_β :

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

In the last step M , the modified scope τ_β replaces τ_α in the original stream τ to form the output τ' (see also Fig. 3). 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:

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

C. Scientific Workflow Challenges Addressed

CHALLENGE 1: Parameter-rich black-box functions. In Virtual Assembly Lines, parameters and inputs to black-box functions are not provided by individual ports nor is there a custom input structure necessary. Instead, data is bound

to the black-box in a two-step process: (1) the read-scope σ defines regions in the input stream where relevant data is located. And (2) via iteration scope ϕ , specific data for each parameter and input can then be selected from the scope or provided in literal form. VAL actors thus exhibit only one input and one output port reducing necessary wiring to a minimum. Of course, the input for a black-box still needs to be specified, and our approach *only moves* the complexity from the graphical wiring into the configurations. However, we believe, that existing mechanism for querying XML-structured data are more suitable for this task than ad-hoc data-structure manipulation and manual wiring.

CHALLENGE 2: Hierarchically organized scientific data. The XML data model of VAL can directly be used to model scientific data. Relationships between data items can easily be maintained by placing them under a common ancestor. As discussed earlier, access to specific parts of the data is provided by XML-selection techniques, for example XPath. On the channels, data is sent in form of XML, and wrappers around the black-boxes can translate to and from the specific (legacy) data requirement of the black-box and XML. Although creating this additional wrapper requires additional work, this needs to be done only once for each black-box and is then re-usable in different workflows. In case a black-box is already invoked as a web service, creating the required SOAP messages is pure XML re-writing, simplifying the creation of a wrapper enormously.

CHALLENGE 3: Flexible grouping of data. In VALs, grouping of data when supplied to the black-boxes is specified by the iteration scope parameter ϕ . We will present a simple extension to XPath in Sect. IV that allows to group selected data based on its location in the XML tree. While other approaches are thinkable in the VAL framework, our suggestion provides a balance between simplicity of use and expressiveness.

CHALLENGE 4: Non-trivial control-flow. Invoking the same black-box multiple times with varying input data can be specified via the iteration scope parameter and thus eliminates shims that were used for managing these loops. Via read scope and iteration scope it can be controlled which data from the input stream is used. Data from the input stream that used to be explicitly routed around an actor is ignored by-default when not inside the read-scope. In VALs, explicit data routing is replaced by the more composition-friendly assembly-line routing, in which data is passed down the line, and actors only pick-up data, they are configured to do so.

CHALLENGE 5: Evolutionary design. A main reason for the added flexibility and change resilience of VALs is that the rigid “is-a” subtype relations in

$$\tau \prec [\alpha] \quad \text{and} \quad [\beta] \prec \tau', \quad (\prec)$$

are relaxed by using read, iteration and write-scope parameters as *built-in* adapters:

$$\phi(\tau_\alpha) \prec [\alpha] \quad \text{with} \quad [\tau_\alpha] = \sigma(\tau)$$

$$\omega([\beta], \tau_\alpha) = \tau_\beta \quad \text{with} \quad M([\tau_\beta], \tau) \prec \tau'$$

In particular, the iteration scope is used to explicitly create the α -typed input for the black-box from read-scope matches. Thus, the actual “ α -data” can be scattered over the read-scope and “packaged” via multiple selections from τ_α or by providing literal values in ϕ . If the data inside the read-scope is not sufficient to construct α -typed data, then the iteration scope can just return an empty sequence and the black-box will not be invoked for this scope match.

Similarly, since most of the XML structure is maintained (only the scope-matches τ_α are modified to τ_β), additional actors in a workflow are less likely to conflict with the following steps. For example, revisiting physical assembly lines, assume worker A does downstream quality control, replacing faulty mirrors with new ones: type τ describes cars, σ selects mirrors of type τ_α . If the glass is broken (broken glass is of type α) then $\phi(\tau_\alpha) = \alpha$ and the worker can get a new glass (fine glass of type β) and replace the old glass to create a fixed mirror $\tau_\beta = \omega(\beta, \tau_\alpha)$. The fixed mirror τ_β changes the overall data product τ only marginally to τ' . Consequently, other workers that, for example, work on wheels or the axles are completely independent. Also, note that by, for instance, selecting only inner and not outer rear view mirrors in σ , the functionality of worker A can be restricted to certain parts of the input.

D. Design and Performance Benefits via Static Analysis

Constructing workflows as virtual assembly lines does not only address the challenges as described above, but also exhibits additional advantages. So can, for example, the output schema of a workflow be predicted via static analysis. Besides benefits during workflow design, VAL workflows can also be executed efficiently, utilizing pipeline or task parallelism or both.

Predicting output schema. VAL workflows have one input and one output port, through which XML data is sent (usually streamed). This makes them ideal for interaction with XML-based data repositories. Furthermore, a well-behaved instance of the framework should have an effective (and ideally efficient) mechanism to solve the following type propagation problem:

$$\tau \xrightarrow{\sigma, \phi, A, \omega} \tau',$$

i.e., given input schema τ and actor configuration σ, ϕ, ω , we want to infer the modified output schema τ' of A . We can then propagate inferred types downstream along any path

$$\tau \xrightarrow{\sigma_1, \phi_1, A_1, \omega_1} \tau_1 \xrightarrow{\sigma_2, \phi_2, A_2, \omega_2} \tau_2 \xrightarrow{\sigma_3, \phi_3, A_3, \omega_3} \dots$$

in a workflow. Once the initial input schema τ is known, the output schema of the workflow can be statically inferred. Because actor configurations σ, ϕ and ω are typically XML queries or XML updates, a large body of previous research can be used as basis for their specification and for type propagation.

Displaying actor dependencies. In contrast to conventional dataflow networks, actors in virtual assembly lines are not tightly coupled to each other via explicit data channels; instead the configurations determine *how* black-boxes are applied to *which* parts of the structure-rich XML stream. While this enables black-boxes to be flexibly applied to data in the input stream without additional wiring, it also hides actor dependencies. Consider for instance an actor A_2 that immediately follows an actor A_1 in a VAL workflow. Unlike in a conventional dataflow network, A_2 does not necessarily depend on the data produced by A_1 (A_2 might work on wheels while A_1 was fixing mirrors).

While constructing and modifying scientific workflows, however, it would be nice to know about the dependencies between actors (just like it was the case in conventional dataflow networks). Removing an actor A_i from a workflow, for example, might be completely fine if A_i 's work is not essential for following actors: To build a car, it is OK to remove an actor that polishes mirrors while it is not OK to remove an actor that installs the axles. In the latter case, all actors that were depending on the existence of axles (adding breaks, adding wheels, pumping up wheels, etc.) are now not able to “do their job” because an essential step (adding axles) is missing. On the other hand, a car might just as well be delivered with slightly dirty mirrors.

In a specific VAL instance, these actor dependencies can be shown explicitly to the workflow developer, if the languages for σ , ϕ , and ω have been chosen carefully. To make these notions more crisp, we will now define the concepts of *actor productivity*, a *required-for* relation between actors, and the notion of *required actors*.

We are generally interested in analyzing composition of actors to larger workflows. A VAL workflow W can be written as $W = A_1, A_2, \dots, A_n$ with A_i being actors each containing configurations σ , ϕ , and ω .

An actor A is **productive** in W if there exists an input to W such that A 's black-box is called. Furthermore, an actor A_i is **required for** another actor A_j in a workflow W (in symbols $A_i \rightsquigarrow A_j$) iff (1) A_j is productive in W , and (2) A_j is not productive in W when A_i has been removed. The inverse relation is called *requires*. In general, an actor A_i is **required** in W if there exists another actor A_j such that A_i is required for A_j .

In our example above, the actor A_1 that adds axles is required (the attach-wheel actor would become unproductive if A_1 were missing), but the actor that only polishes the mirrors is not required.

Execution strategies. A well-chosen instance of the VAL framework should allow for efficient execution of its workflows. Virtual assembly lines do not dictate a particular execution strategy: Based on the characteristics of the black-box functions several approaches for efficient execution are conceivable: (1) Translation to an XML update language and execution inside an XML database, (2) streaming execution with pipeline parallelism, or (3) task-parallel execution, for

example via MapReduce. While we do not want to explore these approaches in detail here, we want to convey the important point that VAL workflows are not only suited for designing scientific workflows, but that it is also possible to execute them efficiently.

(1) *General XML update language.* Since virtual assembly line actors perform updates to XML data, it is natural to explore existing XML update languages as execution backends. A suitable backend would support XML queries and updates, and would be able to call user-defined functions to invoke black-box functions. In Sect. IV-D, we will describe how to “compile” our VAL instance Δ -XML to the XML update language FLUX introduced by Cheney in [11].

(2) *Streaming implementation.* Inspired by Process Networks, one possible execution strategy for virtual assembly lines is streaming. Here, XML data is serialized in a SAX-like manner and sent through (possibly locally distributed) actors token-by-token. To minimize blocking in the individual actors, it should be possible to efficiently recognize read-scope matches τ_α for streaming data. For σ , we will therefore choose XPath expressions that do not have side-axis. In a streaming implementation, actors can work pipeline-parallel on their input data. Also here, it is natural to utilize existing research and software on XML streaming. It is, for example, conceivable to extend FluXQuery [26] with features for XML stream updating to gain an efficient execution engine for VALs.

(3) *Task-parallel implementation.* Virtual assembly lines can also be executed in a task-parallel fashion. If black-box applications account for the most computing costs during workflow execution, than parallelizing their execution is beneficial. VAL workflows naturally provide different granularities for parallelization: A natural choice is to parallelize based on read-scope matches τ_α . If actors are stateless on their read scopes, then ϕ , A , and ω can be applied in parallel over each scope-match produced by σ . The feasibility of such an approach was shown by compiling XML processing pipelines to a series of Map-Reduce programs in [37]. It is thinkable to parallelize even at the granularity of black-box invocations. However, here the right balance between distribution overhead, data shipping and performance gains due to parallelization needs further investigation.

IV. Δ -XML – AN INSTANCE OF THE VAL FRAMEWORK

In this section we propose a Virtual Assembly Line instance called Δ -XML: We specify the used data model, provide concrete syntax and semantic for the VAL actor parameters σ , ϕ and ω , and furthermore define the interface specification for black-box functions. We will reduce workflows written in Δ -XML to the XML update language FLUX [11] and show how the use-cases in Sect. III-D (actor productivity, required relation) can be solved using existing work on type propagation in the FLUX language.

BlackBox ::=	BlackBox: name
	Input: (name of type mod) ⁺
	Output: (name of type mod) ⁺
name ::=	String
type ::=	BaseType
mod ::=	ε *

Fig. 4. Black-box Specification for Δ -XML

A. Δ -XML Data Model

The data model for Δ -XML channels is basically XML with additional types for PCDATA. These types, the **BaseTypes**, are the usual general-purpose types such as *Integer*, *Boolean* and *String*, but also include commonly-used domain-specific types such as *PhylogeneticTree* or *GeneSequence*. Formally, our data model corresponds to rooted, labeled, ordered trees with leaves drawn from BaseTypes. Each node in the tree can have a list of attributes associated with it. An attribute is a name-value pair with names being strings and values being of any BaseType. As usual, each node cannot have two attributes with the same name.

B. Black-box Representation in Δ -XML

We model black-boxes as functions with a list of named input and output parameters, corresponding to the ports in dataflow networks. Each input and each output parameter has an associated name and a type description. As type description, we allow any BaseType, optionally combined with the modifier * to indicate that a list is required as input.

From Δ -XML's perspective, each black-box function can thus be described via a snippet generated from the grammar shown in Fig. 4. Implementation details about how a black-box function is invoked, including any possibly necessary conversion from BaseType to a specific black-box format, are not modeled. This functionality is implemented in the wrapper around the actual black-box code.

C. Δ -XML Configuration Layer

Read scope σ . In Δ -XML, the read scope σ is specified via an XPath expression that uses *child* and *descendent* axes. Since we want to ensure that the selected read-scopes are non-overlapping (according to the general VAL model), we use a *first-match* semantics for the descendent axis //. That means, a breath-first traversal that checks for read-scope matches will not traverse into an already found scope match. While we prohibit general side axis, checking the presence and/or values of attributes attached to nodes along the path is allowed. We present the syntax for our XPath fragment in Fig. 5; the semantics is the standard semantics which selects XML subtrees from the document.

Iteration scope ϕ . The iteration scope parameter is used to invoke the black-box function A and provide them with input data of type α . In Δ -XML, we provide a query (or *binding expression*) for each input parameter of A . Each binding expression can provide data for a single invocation of A , or a set of data that can be used to invoke A multiple times.

ReadScope ::=	ReadScope: XPath
XPath ::=	(Axis Label [AttribTest]?) ⁺
Axis ::=	/ //
Label ::=	String *
AttribTest ::=	exists name name op <i>Literal</i>
	not Test Test and Test Test or Test
op ::=	= <> < > ...

Fig. 5. Read Scope Specification for Δ -XML

Since parameters for black-box functions can themselves be lists, binding expressions select lists of lists. Formally, for each input port i the binding expression B_i represents a query that given the data in the read-scope $r \in \tau_\alpha$ produces a list of lists of values of the base data type associated with port i .

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

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

$$C := B_1(r) \times B_2(r) \times \dots \times B_n(r), \quad (\times)$$

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

Grouping. We suggest to use the standard *foreach* loop with two XPath expressions to select groups:

foreach \$p in XPath₁ **return** XPath₂

To be able to easily grab base-data, we imagine all BaseType-leaf nodes to be implicitly 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 the result sets to form one long output list, instead the result nodes from XPath₂ are grouped by the result of XPath₁, *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. 6. The C data that is available in the read-scope can be selected in different ways: In Fig. 6(a), each C_i is put in a single group each of which will result in a call to the black-box function. In Fig. 6(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. 6(c), whereas in Fig. 6(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 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 programming language here, we believe that binding expressions should be kept rather simple. Using a

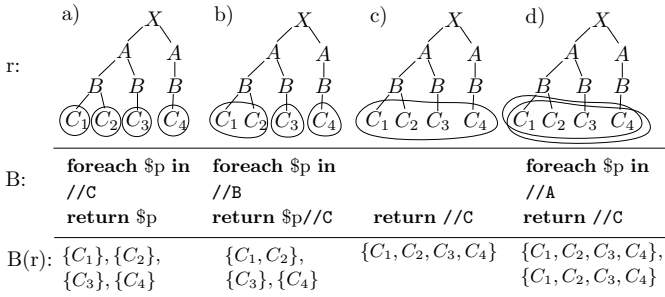


Fig. 6. Example of grouping via binding expression in ϕ .

```

IterationScope ::= Bindings: ( name <- Binding )+
Binding        ::= GXPath | XPath | Literal
GXPath        ::= foreach varname in XPath1 return XPath2
XPath         ::= Anchor (Axis Label Test? )*
Anchor        ::= varname Test? | .
Label         ::= String | *
Axis          ::= / | //
Test          ::= [ Exp ]
Exp           ::= XPath | exists name | name op Literal |
                not Exp | Exp1 and Exp2 | Exp1 or Exp2
op            ::= = | <> | < | > | ...

```

Fig. 7. Iteration Scope Specification for Δ -XML

turing-complete language, would significantly reduce workflow predictability and the effectiveness of static analysis for Δ -XML workflows.

The syntax for the iteration configuration description is given in Fig. 7. Iteration scope is specified for a binding expression for each input parameter of the black-box function. Bindings can either be grouping XPaths, simple XPaths or literals. The **foreach** construct introduces a variable, that can be used as anchor inside XPath₂, the anchor “.” refers to the top of the read-scope match, and can be omitted. Inside the iteration scope, we allow side-tests as part of the XPath expression.

Write scope ω . The purpose of the write scope ω is to insert the results $[\beta]$ of the black-box function A into the read-scope τ_α , or to perform more drastic alternations to the read-scope in order to produce τ_β . An XML query language is not the best candidate here as we often only want to *modify* the scope instead of *creating* a new one from scratch every time. We therefore propose to use an XML update language as basis.

We chose to use the XML update language FLUX [11] to specify the modification on the read-scope τ_α . The FLUX syntax is depicted in Fig. 8. To have access to the results of the black-box, a special variable $\$result$ can be used in the embedded XQuery expressions. For more flexibility while dealing with the black-box results, we not only provide the result value β in $\$result$, but we also allow access to the input parameter α . In particular, each element of the list $\$result$ will contain an XML tree with root node labeled tuple and a subtree for each input and output parameter that was used in an invocation of A . Each subtree is labeled with the name of the parameter and contains the input or output data that was used or created, respectively. It is thus possible

```

Stmt ::= Upd [WHERE Expr]
      | IF Expr THEN Stmt
      | Stmt ; Stmt
      | LET Var := Expr IN Stmt
      | Stmt
Upd  ::= INSERT (BEFORE | AFTER) Path VALUE Expr
      | INSERT AS (LAST | FIRST) INTO Path VALUE Expr
      | DELETE [FROM] Path
      | RENAME Path TO Lab
      | REPLACE [IN] Path WITH Expr
      | UPDATE Path BY Stmt
Path ::= . | Label | node() | text()
      | Path/Path | Path AS Var | Path[ Expr ]

```

Fig. 8. Syntax for FLUX adapted from [11]. Updates as used in write scope ω . “Expr” denote XQuery expressions [11].

to insert the whole $\$result$ -list somewhere into the read scope, or to iterate over the list, and select only specific parameters that should be inserted into the scope.

D. Δ -XML Compilation to FLUX

In the following, we will show how a Δ -XML actor can be translated to FLUX [11], an update language that respects referential transparency and is thus especially suited for static analysis and an efficient implementation.

Necessary FLUX extensions. To compile Δ -XML actors to FLUX programs, FLUX and its type system need to be extended in three ways: (1) adding primitive BaseTypes, (2) adding a construct to call the black-box functions, and (3) adding support for the descendent axis for iteration and read-scope.

Adding BaseTypes. FLUX and the core language LUX, to which FLUX queries are compiled, only contain one primitive type *string*. However, adding BaseTypes to the type system and extending the expression language to reflect the change, does not pose major problems [13], [11].

Adding support to call black-boxes. Cheney proposes type rules for procedures in [11]. Since black-box functions create a number of named output lists from a number of named input lists, with each of the lists containing only BaseTypes, they can easily be incorporated as procedures into the FLUX update language. In the FLUX implementation, control can be delegated to the wrappers to interface with the actual black-boxes. We will use the black-box function name inside the body of a let statement to denote the function call (see Fig. 10 line 9). Input parameters are provided in parenthesis output parameters are bound to the output values inside the let statement. For ease of presentation, parameters are matched by position.

Adding support for descendent axes. FLUX does not allow the use of descendent axis to avoid overlapping selections for the *focus* of an update. Descendent operators in the read-scope are defined to use a *first-match* semantics to prevent overlapping scope matches. When compiling FLUX to LUX (as it is done in [11]) it is therefore possible to rewrite a //

```

1 BlackBox: CipresTreeInference
2 Input:  method of String
3          geneSequences of GeneSequence*
4          seed of Float
5          maxIterations of Integer
6 Output: tree of PhyloTree*
7          quality of Float
8 ReadScope: //Species
9 Bindings:
10 method <- foreach $p in //Method return $p/String
11 geneSequences <- return //Alignment//GeneSequence
12 seed <- {42}, {23}
13 maxIterations <- return /MaxIteration/Integer
14 WriteScope:
15 INSERT AS LAST INTO . VALUE Trees[ $result ]

```

Fig. 9. Example for Δ -XML actor configuration

operator into a procedure that exactly implements the first-match semantics. Descendent operators in the iteration scope are not used to select input focus and are thus already allowed in FLUX because they are part of μ XQ (doS-operator) [13], which is the XQuery language used in FLUX.

Rewriting Δ -XML to FLUX. A Δ -XML workflow $W = A_1, \dots, A_n$ is compiled to a FLUX program F by rewriting each Δ -XML actor A_i into FLUX statements f_i that are then stringed together in the order of the original actors:

$$W = A_1, \dots, A_n \rightsquigarrow f_1; \dots; f_n = F$$

We will explain the transformation based on the example Δ -XML actor in Fig. 9. In the compiled version (Fig. 10) FLUX update statements, as defined in Fig. 8, are written in capital letters whereas statements from the query language μ XQ are written with small letters. As shown in Fig. 10 line 1, each actor is transformed into one **UPDATE .. AS .. BY** statement; consequently the update given after **BY** is performed on each result returned by the read scope (here //Species). Additionally, the current read scope is bound to the variable \$readS. In the **LET**-statement (line 2), the result-list \$result is created. For each input parameter of the black-box, a variable (e.g., \$method) is introduced. If the binding was given via a grouping XPath expression (line 10 in Fig. 9), a fresh variable (here \$methodGrp) is used in a for loop to iterate over the first path; the second path (here \$p/String) is adjusted if it refers back to the variable \$p (here it is replaced by \$methodGrp). In case the binding was a non-grouping XPath expression in the actor (Fig. 9 line 11), the variable (here \$geneSequences) is bound via a simple **let**-statement (Fig. 10 line 4). For literal values, **for**-loops are introduced if additional groups have been indicated via {...} (Fig. 9 line 12 and Fig. 10 line 6), otherwise a **let**-statement is used. If a non-list parameter is bound with a simple **let**-statement, originating from a non-grouping binding (as in Fig. 9 line 13), an additional **if**-statement is used to only call the black-box function if the parameter was bound (line 7). Without this **if**-statement, the FLUX type system would not type-check the program as the black-box procedure could possibly be called with the value “()” for the empty sequence. As body of all nested **loop** and **let**-statements, the black-box function is called and provided with the input parameters. The

```

1 UPDATE //Species AS $readS BY {
2 LET $result :=
3 for $methodGrp  $\in$  $readS//Method return
4 let $method := $methodGrp/String in
5 let $geneSequences := $readS//Alignment//GeneSequence in
6 for $seed  $\in$  (42, 23) return
7 if ($readS/MaxIteration/Integer) then
8 let $maxIterations := $readS/MaxIteration/Integer return
9 let ($tree, $quality) = CipresTreeInference(
10 $method, $geneSequences, $seed, $maxIterations) in
11 tuple[method[$method], geneSequences[$geneSequences],
12 seed[$seed], maxIterations[$maxIterations],
13 tree[$tree], quality[$quality] ]
14 else ()
15 IN
16 IF ($result) THEN
17 INSERT AS LAST INTO . VALUE Trees[ $result ] }

```

Fig. 10. FLUX-Code corresponding to Δ -XML actor given in Fig. 9.

output values (or list of values) are bound to the variables (line 9). Then (lines 11-14), the result tuple is created with subtrees that are labeled with the names of input and output parameters and which contain the corresponding data. The write-scope statement, which will update the scope τ_α if the black-box function was called (and thus \$result is not empty) is then pasted in line 18.

E. Static Analysis for Δ -XML Workflows

Once a Δ -XML workflow has been compiled into FLUX programs, static analysis techniques available for FLUX programs can readily be used to provide additional benefits via static analysis. In particular, during compile time we can (1) guarantee that all black-box functions are provided with the correct base data, (2) predict the schema of workflow output, (3) display actor dependencies, and (4) split-up the work for an efficient parallel execution.

Type safety. Using the FLUX type-system, we can verify *before* the workflow is run that all binding expressions will select data compatible with the black-box functions. This can be done by adding a type declaration Δ for each black-box function and by simply type-checking the FLUX program $f_1; f_2; \dots; f_n$. The typing rule for procedures (which use the declarations Δ (see [11]) ensures that black-boxes will be called with compatible base-data only.

Output schema prediction. Given a specific input schema (or the Any-type) as input, we can make use of FLUX’s type system and predict the output schema of the workflow by simply applying the rules given in [11].

Actor Dependencies. The basis for detecting unproductive actors and actor dependencies is the dead-code analysis available for FLUX and μ XQ. Dead-code analysis for FLUX [11] is an extension of the *path-error* analysis for μ XQ described in [14]. In FLUX, the analysis detects subexpressions (or statements) that are equivalent to the operation *skip*, i.e., that do not change the input data. For query-expressions in μ XQ the analysis finds expressions that are equal to the empty

```

INPUT:  $F = f_1, f_2, \dots, f_n$    OUTPUT: required-for-relation R
for  $i := 1$  to  $n$  do
   $F' := \text{RemoveActor}(F, i)$ 
   $\text{turned-unprod} := \text{NotProdIn}(F') \cap \text{ProdIn}(F)$ 
  foreach  $j \in \text{turned-unproductive}$  do  $R.\text{Insert}(i, j)$ 
return R

```

Fig. 11. Generating the *required-for* relation R of actors in a workflow that has been compiled to a FLUX program F. The code uses the productivity checks `NotProdIn(..)` and `ProdIn(..)` available for FLUX programs. `ProdIn(X)` returns the set of productive actors in X, `NotProdIn(X)` returns the not productive actors.

sequence `()`. Examples for dead-code in FLUX is a **for loop** ranging over a path that will not have any bindings, or an **UPDATE Path BY** statement, in which the *Path* will always evaluate to an empty list. We can therefore use the algorithm in [11] to detect cases in which no read-scope match will occur.

Furthermore, the black-box function will not be called if one of the non-list parameters is not provided any data. In case the parameter is filled with a simple XPath expression, the **if** statement guarding the **let** binding for the variable will not be satisfied. If the path is filled with a for-loop no data will be available to be looped over and the black-box function is not be invoked either.

Whenever the black-box function is not called, `$result` will be empty and no update will be performed. However, FLUX’s rule for its **IF** statement only detects the if-statement as unproductive if both alternatives are unproductive. Since μXQ can analyze emptiness of variables [14], we can slightly improve FLUX’s analysis to Additionally mark the **IF** statement unproductive if the current type of the expression is the empty sequence and the else branch is unproductive (see appendix for the necessary rule changes).

With this slightly modified FLUX analysis, we can detect unproductive actors A_i in Δ -XML workflows by checking the associated FLUX statements f_i . To analyze actor dependencies in a workflow W , we would simply check which actors cause other actors to turn unproductive if removed. To generate the whole “required-for” relation as defined in Sect. III-D, the pseudo-code in Fig. 11 can be used. The required-for relation, can be displayed to the user when integrating multiple actors in a workflow. With this information, the developer can verify that there are no typos in the XPath expressions (as otherwise actors would be unproductive). More importantly, this information also provides feedback on which actors are essential for downstream steps (and should thus not be removed).

Parallel execution strategies. The FLUX semantics guarantees that no update can change data outside its current focus. Since all Δ -XML actors are compiled as updates having (at least) the read scope as focus, workflows can be executed in a streaming fashion, utilizing pipeline parallelism over the read-scope. Furthermore for reasons of simplicity and repeatability, black-box functions are usually stateless, *i.e.*, their output does not depend on previous invocations but only on the input. Then, the parallelization techniques presented in [37] are applicable.

V. DISCUSSION AND RELATED WORK

We have presented a new model of computation based on *virtual assembly lines*, which provides an abstract modeling and design paradigm for process pipelines. Compared with other approaches [1], virtual assembly lines can yield more modular, change-resilient process networks, mainly due to the adoption of an assembly line metaphor applied to actors with a configurable “work scope”. In essence, the black-box function inside an actor is not directly fed via input ports, but a flexible configuration layer consisting of read, iteration and write-scope is used. Furthermore, Δ -XML actors largely maintain the structure of the input stream and only modify the stream within their scope. Since we use the well-behaved XML-Update language FLUX [11] to implement Δ -XML actors, our actors can be statically analyzed. This allows for additional features such as guaranteeing type-safety for Back-box function calls, predicting the output schema of a workflow, and revealing actor dependencies. Earlier work on XML processing pipelines [38], [37] show the computational feasibility of the approach.

Static analysis is based on the work on FLUX [11], μXQ [14], and earlier work on regular expression types [20], [3]. Our work heavily utilizes research on FLUX and its type system. As we have shown, Δ -XML actors can be programed in the FLUX language (with few extensions as described in this work). In addition to existing research on FLUX, we focus on providing an abstraction for creating large FLUX programs from smaller ones, *i.e.*, to construct workflows using actors as components. We show how actors with scopes (*i.e.*, certain FLUX programs) facilitate robust and change-resilient workflows with simple wiring, as well as improve the workflow design process via providing feed-back about unproductivity.

Recent years have seen a surge of interest in the general area of scientific workflows, including foundations, methods, systems, and applications, *e.g.*, see [19], [32], [34] or [28] for a survey. The idea of employing an assembly line paradigm to improve process network designs and/or performance is not new. Most closely related to the present work is the collection-oriented modeling and design (COMAD) approach [30], [29] for scientific workflows. In particular, Δ -XML extends COMAD by (1) presenting a formal model to clearly define read, iteration and write-scope, and (2) by showing how Δ -XML actors can be compiled to FLUX programs that then allow static analysis.

Assembly-line processing is also related to *flow-based programming*, an approach that also provides improved dataflow (workflow) reusability and “configurable modularity” [31], but which does not include our extensions (actor scopes and static analysis). Other approaches to improve workflow designs are based on adapters or shims [5], [6], [34], [21] or consider ways of combining dataflow and control-flow [19], [7]. Dataflow and control-flow are “married” in the basic process network model [23], [27], which can lead to simple and elegant designs for simple data-driven computations, but also to rather complex

and change-sensitive (*i.e.*, “brittle”) designs, when control-flow is very different from dataflow [7].

There is also a significant amount of work in the area of query processing over XML streams, *e.g.*, see [8], [9], [2], [25], [18], [10] among many others. Most of these approaches consider optimizations for specific XML query languages or language fragments, sometimes taking into account additional aspects about the streaming data (*e.g.*, sliding windows). To the best of our knowledge, our approach is unique in that it shows how virtual assembly lines and Δ -XML can address the unique challenges of designing scientific workflows.

REFERENCES

- [1] I. Altintas, E. Jaeger, C. Berkley, M. Jones, B. Ludäscher, and S. Mock. Kepler: An extensible system for design and execution of scientific workflows. In *16th SSDBM*, Santorini, Greece, 2004.
- [2] M. Balazinska, H. Balakrishnan, S. Madden, and M. Stonebraker. Fault-Tolerance in the Borealis Distributed Stream Processing System. In *SIGMOD*, 2005.
- [3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *Intl. Conf. on Functional Programming (ICFP)*, pages 51–63, New York, NY, USA, 2003.
- [4] BLAST: Basic Local Alignment Search Tool. <http://www.ncbi.nlm.nih.gov/BLAST>, 2009.
- [5] S. Bowers and B. Ludäscher. An ontology driven framework for data transformation in scientific workflows. In *International Workshop on Data Integration in the Life Sciences (DILS)*, 2004.
- [6] S. Bowers and B. Ludäscher. Actor-oriented design of scientific workflows. In *ER*, 2005.
- [7] 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.
- [8] 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 *CIDR*, 2003.
- [9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, pages 379–390, 2000.
- [10] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. In *ICDE*, 2006.
- [11] J. Cheney. FLUX: functional updates for XML. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*, pages 3–14, New York, NY, USA, 2008. ACM.
- [12] Cipes: Cyberinfrastructure for phylogenetic research. <http://www.phylo.org/>.
- [13] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for path correctness of XML queries. In *Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 126–137. ACM New York, NY, USA, 2004.
- [14] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Static analysis for path correctness of xml queries. *J. Funct. Program.*, 16(4-5):621–661, 2006.
- [15] J. Felsenstein. PHYLIP (phylogeny inference package) version 3.6. *Distributed by the author: Department of Genome Sciences, University of Washington, Seattle*, 2004.
- [16] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Scientific and Statistical Database Management, 2002. Proceedings. 14th International Conference on*, pages 37–46, 2002.
- [17] J. Freire, C. Silva, S. Callahan, E. Santos, C. Scheidegger, and H. Vo. Managing rapidly-evolving scientific workflows. *Lecture Notes in Computer Science*, 4145:10, 2006.
- [18] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. *TODS*, 29(4):752–788, 2004.
- [19] 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.
- [20] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *TOPLAS*, 2005.
- [21] 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.
- [22] 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.
- [23] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Proc. of the IFIP Congress 77*, pages 993–998, 1977.
- [24] Kepler project. <http://kepler-project.org>.
- [25] 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*, 2004.
- [26] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. FluXQuery: An Optimizing XQuery Processor for Streaming XML Data. *VLDB*, pages 1309–1312, 2004.
- [27] E. A. Lee and T. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–799, May 1995.
- [28] B. Ludäscher and C. A. Goble. Guest editors’ introduction to the special section on scientific workflows. *SIGMOD Record*, 34(3), 2005.
- [29] T. McPhillips, S. Bowers, and B. Ludäscher. Collection-oriented scientific workflows for integrating and analyzing biological data. *DILS*, 2006.
- [30] T. M. McPhillips and S. Bowers. An Approach for Pipelining Nested Collections in Scientific Workflows. *SIGMOD Record*, 34(3):12–17, 2005.
- [31] J. P. Morrison. *Flow-Based Programming – A New Approach to Application Development*. Van Nostrand Reinhold, 1994.
- [32] 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.
- [33] 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)*, Monterey Bay, June 2007.
- [34] 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.
- [35] 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.
- [36] Taverna. <http://taverna.sourceforge.net>.
- [37] D. Zinn, S. Bowers, S. Köhler, and B. Ludäscher. Parallelizing XML Processing Pipelines via MapReduce. Technical Report CSE-2009-12, UC Davis, 2009.
- [38] D. Zinn, S. Bowers, T. M. McPhillips, and B. Ludäscher. X-CSR: Dataflow Optimization for Distributed XML Process Pipelines. In *ICDE*, pages 577–580, 2009.

APPENDIX

A Slightly More precise Productivity Check for If Statements.

We follow the in XQuery common convention that the empty sequence “()” evaluates to `false`, if used in if-then-else expressions. We can thus improve the dead-code analysis for unproductive if-statements by replacing the generic rule (1):

$$\frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash^\alpha \{ \tau \} (s_1)_{l_1} \{ \tau_1 \} \& L_1 \quad \Gamma \vdash^\alpha \{ \tau \} (s_2)_{l_2} \{ \tau_2 \} \& L_2}{\Gamma \vdash^\alpha \{ \tau \} (\text{if } e \text{ then } (s_1)_{l_1} \text{ else } (s_2)_{l_2})_l \{ \tau_1 | \tau_2 \} \& (L_1 \cup L_2) [l_1, l_2 \Rightarrow l]} \quad (1)$$

with the following two rules:

$$\frac{\Gamma \vdash e <: () \quad \Gamma \vdash^\alpha \{ \tau \} (s_2)_{l_2} \{ \tau_2 \} \& L_2}{\Gamma \vdash^\alpha \{ \tau \} (\text{if } e \text{ then } (s_1)_{l_1} \text{ else } (s_2)_{l_2})_l \{ \tau_2 \} \& (L_2) [l_2 \Rightarrow l]} \quad (2)$$

$$\frac{\Gamma \vdash e \not\prec: () \quad \Gamma \vdash^a \{\tau\}(s_1)_{l_1} \{\tau_1\} \& L_1 \quad \Gamma \vdash^a \{\tau\}(s_2)_{l_2} \{\tau_2\} \& L_2}{\Gamma \vdash^a \{\tau\}(\text{if } e \text{ then } (s_1)_{l_1} \text{ else } (s_2)_{l_2})_l \{\tau_1 | \tau_2\} \& (L_1 \cup L_2)[l_1, l_2 \Rightarrow l]} \quad (3)$$

While the rule (1) determines the if-then-else-statement unproductive only if both sides are unproductive ($l_1, l_2 \Rightarrow l$), the new version (2) checks whether the type of e is $()$, and if so, infers a tighter result type for the update (τ_2 instead of $\tau_1 | \tau_2$) and marks the if-then-else-statement unproductive already if the second statement was unproductive ($l_2 \Rightarrow l$). In case the type of e is not the empty sequence, then the old rule is used (3) because a type $\tau \neq ()$ does not guarantee that all its value are non-empty; consider for example the following type $\tau = \text{true} \mid ()$, which has an empty and a non-empty instance.