XML-Based Computation for Scientific Workflows

Daniel Zinn [#1], Shawn Bowers [*2], Bertram Ludäscher [#3]

[#]*Department of Computer Science, University of California at Davis*
1 Shields Ave, Davis, CA, USA
[1]dzinn@ucdavis.edu
[3]ludaesch@ucdavis.edu

[*]*Department of Computer Science, Gonzaga University*
502 East Boone Avenue, Spokane, WA, USA
[2]bowers@gonzaga.edu

***Abstract*— Scientific workflows are increasingly used to rapidly integrate existing algorithms to create larger and more complex programs. However, designing workflows using purely dataflow-oriented computation models introduces a number of challenges, including the need to use low-level components to mediate and transform data (so-called *shims*) and large numbers of additional "wires" for routing data to components within a workflow. To address these problems, we employ *Virtual Data Assembly Lines (VDAL)*, a modeling paradigm that can eliminate most shims and reduce wiring complexity. We show how a VDAL design can be implemented using existing XML technologies and how static analysis can provide significant help to scientists during workflow design and evolution, e.g., by displaying actor dependencies or by detecting so-called unproductive actors.**

I. Introduction

Scientists are often faced with the problem of combining different software components to form larger computational workflows (e.g. data analysis pipelines). Scientific workflow systems have recently been proposed as a general approach for helping scientists with these component integration tasks. For example, Taverna [5] and Kepler [3] both allow users to build workflows that combine locally available programs, which might be written in different languages, with programs that are accessed via web services. Once an existing algorithm has been wrapped as a component (*actor*) in these systems, it can interoperate with other actors without manual intervention—the details concerning its invocation are hidden from the domain scientist, allowing them to instead focus on defining the desired workflow.

Many systems follow a dataflow-oriented approach: computational steps are represented as nodes (actors) connected via *channels* in a dataflow graph. The scientist then builds more complex analyses by placing actors on a canvas and connecting them using a scientific workflow design tool. Despite this abstraction, it is currently still hard to construct complex workflows [4]. In particular, dataflow-oriented approaches can lead to overly complex workflow graphs due to a number of workflow design challenges [7], [6]: (i) parameter-rich functions and services, (ii) maintenance of data cohesion, (iii) conditional execution, (iv) iterations over cross products, and (v) workflow evolution. In particular, workflows tend to have many channels and thus very complex workflow graphs. Furthermore, 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*) can further increase the complexity of a workflow, resulting in workflows that are hard to construct, extend, and maintain. As an example, consider a purely dataflow-oriented workflow as in Figure 1(a): besides having scientifically meaningful actors (here: `A`, `B`, and `C`) there are additional "shim" actors (shown as black boxes) that are only necessary for implementing control-flow (e.g., to iterate over data) and for maintaining data associations.

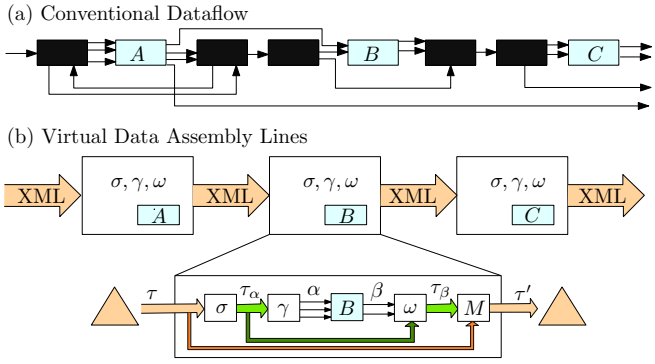

Fig. 1. In Virtual Data Assembly Lines, data transformation is moved to a configuration layer, denoted by α, γ, and ω, to reduce wiring complexity and facilitate more straightforward workflow designs with re-usable components.

Contributions. In this paper, we extend our work in [7], which introduces the general VDAL framework for modeling scientific workflows: We **(1)** propose a specific instance of the VDAL model, called Δ-XML; **(2)** show how to compile Δ-XML models to the XML update language FLUX [1]; and **(3)** demonstrate how the FLUX type system can be used to provide additional features for scientific workflow designers, such as actor-dependency analysis.

II. The VDAL Paradigm

In a *Virtual Data Assembly Line (VDAL)*, ad-hoc data manipulation as it is performed by shims in the conventional dataflow design is moved to a set of configurable components $(\sigma, \gamma, \omega, M)$ around the scientific actor as shown in Fig. 1(b). Furthermore, data is organized as an XML stream and thus

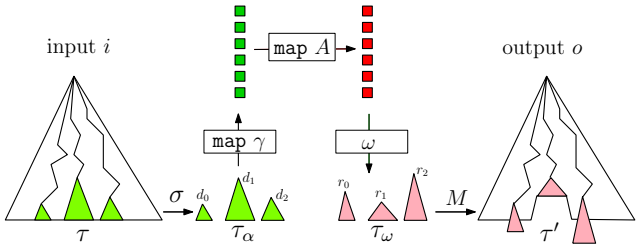

Fig. 2. Dataflow inside VDAL Actor.

associations between data can be maintained as part of the XML structure. Each of the components σ, γ, ω, and M has a specific purpose for the transformation of the data as it flows through the actor (see Fig. 2): The *scope* σ partitions the incoming data stream into work-pieces $d_1, \ldots, d_n$; the *input assembler* γ takes each of the d_i and selects appropriate input datasets (depicted as green squares) to invoke the scientific function A on each set. The output data from A (red squares) is then inserted into the currently processed scope d_i to form the modified *result* r_i. The merger component M then simply places each r_i into the data stream at the position of its corresponding d_i. In practice, updating the scopes d_i to r_i often happens *in place* as the data streams through the actor, so the final merge step M is implicit in this case.

In contrast to the conventional approach, where shim actors have to be placed and connected, or even custom-written, to select input data and to create invocation lists for the scientific actor A, the VDAL workflow designer only needs to configure the components σ, γ, and ω of a VDAL actor. This approach has many advantages [7]: (1) Configurations are more declarative and thus describe *what* data should be selected and *where* it should be placed as opposed to an operational description of *how* data is selected, assembled and dissembled. (2) Since the configurations refer to labels in the XML stream, the processing logic is decoupled from earlier actors. As a consequence, a VDAL actor is oblivious to how and by which actor the data was created and put into place in its input stream. (3) Furthermore, configurations can be chosen from a restricted language and can thus allow the workflow system itself to reason about the worflow as a whole to provide valuable modeling support for the developer. This is not possible in the conventional approach where shim actors are considered black-boxes, since they are often written in a general-purpose programing language, which makes their analysis hard or impossible.

III. Δ-XML – An Instance of VDAL

In this section we propose Δ-XML, an instance of Virtual Data Assembly Lines. We specify the Δ-XML data model, illustrate syntax and semantics of actor configurations σ, γ and ω, and define the interface specification for actors that implement the underlying scientific functions.

A. Δ-XML Data Model

The data model for the Δ-XML channels is XML with additional types for `CDATA`. 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*. Thus, our data model corresponds to rooted, labeled, ordered trees. We differentiate between *collection nodes* (short: *collections*) and *base data* nodes. A data node is labeled with the name of a BaseType and contains a data value of that type; data nodes can only occur as leaves in the XML tree. Collections are labeled and can occur as inner nodes (containing other collections or data) or as leaves (empty collections). Collection and data nodes can have attribute lists associated with them. An attribute is a name-value pair, where names are strings and values are of any BaseType. As usual, a node can have at most one attribute for any given attribute name.

B. Scientific Actor Representation in Δ-XML

Scientific actors wrap existing algorithms, tools, and services, and have associated lists of inputs and outputs, corresponding to the ports in pure dataflow networks. Each input and output parameter has an associated name and type. The type T* denotes a list of values whose elements are of type T.

C. Δ-XML Configuration Layer

We now describe the configuration parameters σ, γ, and ω and provide an illustrative example; for details see [6].

Scope σ. In Δ-XML, the scope σ is specified via an XPath expression that uses *child* and *descendant* axes. Since we want to ensure that scope-matches d_i are non-overlapping, we use a *first-match* semantics for the descendant 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 axes, checking the presence and values of attributes attached to nodes along the path is allowed.

Input assembler γ. The input assembler is used to invoke the scientific actor A and provide it with input data. We use 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. Since parameters for scientific 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 scope d_i produces a list of lists of values of the base data type associated with port i. The scientific actor A is then invoked once for each element of the Cartesian product:

$$B_1(d_i) \times B_2(d_i) \times \cdots \times B_n(d_i) \qquad (\times)$$

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

foreach $\$p$ **in** XPath_1 **return** XPath_2

Here, selecting data 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 XPath_2 are grouped

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

Fig. 3. Example for Δ-XML actor configuration

by the result of XPath_1, i.e., for each new node bound to p a new group is formed.

Write expression ω. The purpose of ω is to insert the results of the scientific actor A into the scope d_i, or to make other changes within the scope. We chose to use the XML update language FLUX [1] for ω. To have access to the results of A, a special variable `$result` is used in the FLUX expressions. Additionally, for each result tuple we allow access to the input data of A that produced it. In particular, each element of the list `$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.

VDAL Actor Example. In Fig. 3, the configuration for an `CipresTreeInference` actor is shown. The scientific actor has four input parameters, and produces a list of phylogenetic trees as output. The actor's scope is //Species, such that input data is searched for only within subtrees labeled with `Species`. The service should be called for each method that is under a `Method` collection in the scope with seeds 23 and 42 each. Gene sequences are read from the scope under an `Alignment` collection; the MaxIterations parameter from inside the `MaxIteration` collection. The output list of resulting trees is inserted within a new subtree labeled `Trees` inside the current scope.

D. Δ-XML Compilation to FLUX

To compile Δ-XML actors to FLUX programs, FLUX and its type system need to be extended in three ways:

(1) Adding BaseTypes. FLUX only contains 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 [2], [1].

(2) Adding support to call scientific actors. Cheney proposes type rules for procedures in [1]. Since scientific actors 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 FLUX. Also, the FLUX implementation can easily be extended by delegating control to the wrappers when scientific actors are called. We will use the service name inside the body of a **LET**-statement to denote the function call (see Fig. 4 line 9). Input parameters are provided in parentheses; output parameters are bound to the output values inside the **LET**-statement. For ease of presentation, parameters are matched by position.

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

Fig. 4. FLUX program corresponding to Δ-XML actor given in Fig. 3.

(3) Adding support for descendant axes. FLUX does not allow the use of descendant axis to avoid overlapping selections for the *focus* of an update. Descendant operators in VDAL scopes are defined to use a *first-match* semantics to prevent overlapping scope matches. When compiling FLUX to LUX (as it is done in [1]) it is therefore possible to rewrite a // operator into a procedure that exactly implements the first-match semantics. Descendant operators in the input assembler are not used to select input focus and are thus already allowed in FLUX because they are part of μXQ (`dos`-operator) [2], which is used as a sub-language in FLUX.

Rewriting Δ-XML to FLUX. A Δ-XML workflow $W = A_1 \rightarrow \ldots \rightarrow A_n$ 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:

$$W = A_1 \rightarrow \ldots \rightarrow A_n \;\rightsquigarrow\; f_1; \ldots; f_n = F$$

We now explain the transformation of the example actor in Fig. 3 to the FLUX code in Fig. 4. As shown in Fig. 4 line 1, each actor is transformed into one **UPDATE** .. **AS** .. **BY** statement; thus the update given after **BY** is performed on each result returned by the scope (here: //Species). Additionally, the current scope is bound to the variable `$readS`. In the **LET**-statement (line 2), the result-list `$result` is created. For each input parameter, a variable (e.g., `$method`) is introduced. If the binding was given via a grouping XPath expression (line 10 in Fig. 3), a fresh variable (here `$methodGrp`) is used in a **FOR**-loop to iterate over the first path; the second path (`$p/String`) is adjusted if it refers back to the variable `$p` (here it is replaced by `$methodGrp`). In case the binding was a non-grouping XPath expression in the actor (Fig. 3 line 11), the variable (here: `$geneSequences`) is bound via a simple **LET**-statement (Fig. 4 line 4). For literal values, **FOR**-loops are introduced if additional groups have been indicated via {...}

(Fig. 3 line 12 and Fig. 4 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. 3 line 13), an additional **IF**-statement is used to only call the scientific function if the parameter was bound (line 7). Without this **IF**-statement, the FLUX type system would not type-check the program as the scientific procedure could possibly be called with the value "()" for the empty sequence. As body of all nested **FOR** and **LET**-statements, the scientific function is called and provided with the input parameters. The 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-expression statement, which will update the scope if the scientific 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.

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 scientific functions. This can be done by adding a type declaration for each scientific function and by simply type-checking the FLUX program $f_1; f_2; \ldots; f_n$. The typing rule for procedures (see [1]) ensures scientific functions are 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 [1].

Actor Dependencies. The basis for detecting unproductive actors and actor dependencies is the dead-code analysis available for FLUX and μXQ. Dead-code analysis for FLUX [1] is an extension of the *path-error* analysis for μXQ described in [2]. The analysis detects subexpressions that do not change the input data. For query-expressions in μXQ the analysis finds expressions that are equal to the empty sequence ( ). An example 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 [1] to detect cases in which no scope match will occur. Furthermore, the scientific actor will not be called if one of the non-list parameters is not provided with 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 scientific function is not invoked either.

Whenever the scientific function is not called, `$result` will be empty and no update will be performed. However, FLUX's rule for its **IF** statement only detects it as unproductive if both alternatives are unproductive. Since μXQ can analyze emptiness of variables [2], we can slightly improve FLUX's analysis and also mark the **IF** statement unproductive whenever the current type of the expression is the empty sequence and the else branch is unproductive (see Appendix). 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 and obtain a new *required for* relation. This relation can be displayed to the user when integrating multiple actors in a workflow, and the designer can thus 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.

ACKNOWLEDGMENT

We thank Timothy McPhillips who developed and impelemented COMAD (Collection Oriented Modeling & Design), which VDAL is an abstraction of. This work supported in part by NSF IIS-0612326, OCI-0722079, DBI-0619060, IIS-0630033, DOE DE-FC02-07ER25811.

REFERENCES

[1] J. Cheney. FLUX: Functional updates for XML. In *ICFP*, pages 3–14, 2008.

[2] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for path correctness of XML queries. In *ICFP*, pages 126–137, 2004.

[3] Kepler project. http://kepler-project.org.

[4] T. McPhillips, S. Bowers, D. Zinn, and B. Ludäscher. Scientific Workflow Automation for Mere Mortals. *FGCS*, 25(5):541–551, 2009.

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

[6] D. Zinn, S. Bowers, and B. Ludäscher. XML-Based Computation for Scientific Workflows. Technical Report CSE-2009-21, UC Davis, 2009.

[7] D. Zinn, S. Bowers, T. McPhillips, and B. Ludäscher. Scientific workflow design with data assembly lines. In *WORKS*, 2009.

APPENDIX

A Slightly More Precise Productivity Check for IF Statements

We follow the common XQuery 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):

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

with the following two rules:

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

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

While the rule (1) determines the if-then-else-statement unproductive only if both sides are unproductive ($l1, l2 \Rightarrow l$), 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 $\tau =$`true` | (), which has an empty and a non-empty instance.