Actor-Oriented Design of Scientific Workflows*

Shawn Bowers[†] Bertram Ludäscher[†‡]

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

Abstract. Scientific workflows are becoming increasingly important as a unifying mechanism for interlinking scientific data management, analysis, simulation, and visualization tasks. Scientific workflow systems are problem-solving environments, supporting scientists in the creation and execution of scientific workflows. While current systems permit the creation of executable workflows, conceptual modeling and design of scientific workflows has largely been neglected. Unlike business workflows, scientific workflows are typically highly data-centric naturally leading to dataflow-oriented modeling approaches. We first develop a formal model for scientific workflows based on an actor-oriented modeling and design approach, originally developed for studying models of complex concurrent systems. Actor-oriented modeling separates two modeling concerns: component *communication* (dataflow) and overall workflow *coordination* (orchestration). We then extend our framework by introducing a novel *hybrid* type system, separating further the concerns of conventional data modeling (*structural data type*) and conceptual modeling (*semantic type*). In our approach, semantic and structural mismatches can be handled independently or simultaneously, and via different types of *adapters*, giving rise to new methods of scientific workflow design.

1 Introduction

Scientific workflows are quickly becoming recognized as an important unifying mechanism to combine scientific data management, analysis, simulation, and visualization tasks. Scientific workflows often exhibit particular traits, e.g., they can be data-intensive, compute-intensive, analysis-intensive, and visualization-intensive, thus covering a wide range of applications from low-level "plumbing workflows" of interest to Grid engineers, to high-level "knowledge discovery workflows" for scientists [11]. Consequently, workflows steps can have very different granularities and may be implemented as shell scripts, web services, local application calls, or as complex subworkflows.

A *scientific workflow system* is a problem-solving environment that aims at simplifying the task of "gluing" these steps together to form executable data management and analysis pipelines. While current systems permit the creation of executable workflows, conceptual modeling and design of scientific workflows has been largely neglected. Unlike business workflows, scientific workflows are typically highly data-centric, naturally

* This work supported in part by NSF/ITR 0225673 (GEON), NSF/ITR 0225676 (SEEK), NIH/NCRR 1R24 RR019701-01 (BIRN-CC), and DOE DE-FC02-01ER25486 (SDM).

leading to dataflow-oriented modeling approaches, while business workflow modeling is dominated by control, event, and task-oriented approaches [17], making them less suitable for the modeling challenges of scientific workflows.

This paper addresses three important problems in scientific-workflow design and engineering. First, in existing systems it is often unclear what constitutes a scientific workflow, and there are few if any abstract models available to describe scientific workflows. (By abstract model, we mean a model for scientific workflows analogous to data models in database management.) Second, existing systems do not support the end-to-end development of scientific workflows, in particular, design methods and frameworks for the early stages of conceptual design do not exist. And third, in scientific workflow systems such as KEPLER [11] that aim at providing a unified environment where workflows and their components can be shared and reused, mechanisms do not exist that support the discovery, reuse, and adaptation of existing workflows and components.

To address these issues, we first develop a *formal model for scientific workflows* (Section 3) based on an actor-oriented modeling approach, originally developed for studying complex concurrent systems [9]. A benefit of actor-oriented modeling is that it separates two distinct modeling concerns: component *communication* (dataflow) and overall workflow *coordination* (a.k.a. orchestration). We then extend this framework by introducing a novel *hybrid type system*, separating further the concerns of conventional data modeling (*structural data type*) and conceptual modeling (*semantic type*). The separation of types facilitates the *independent* validation of structural and semantic type constraints and offers a number of benefits for scientific workflow design and component reuse. Structural and semantic types can also be explicitely *linked* in our approach, using special (hybridization) constraints. These constraints can be exploited in various ways, e.g., to further propagate and refine known (structural or semantic) types in scientific workflows, or to infer (partial) structural mappings between structurally incompatible (but semantically compatible) workflow components.

Based on our formal model, we also introduce a number of basic *modeling primitives* that a workflow designer can apply to evolve a formal scientific workflow design in a stepwise, controlled manner (Section 4). The different modeling primitives give rise to distinct *design strategies*, including task-driven vs. data-driven, structure-driven vs. semantics-driven, and top-down vs. bottom-up. Two important design primitives are *actor replacement* and *adapter insertion*. Both primitives, when combined with the hybrid type system, yield powerful new component discovery and adaptation mechanisms.

2 Preliminaries: Business vs. Scientific Workflows and KEPLER

The characteristics and requirements of scientific workflows partially overlap those of business workflows. Historically, business workflows have roots going back to office automation systems, and more recently gained momentum in the form of business process modeling and business process engineering [2,16,18]. Today we see influences of business workflow standards in web-service choreography standards. Examples include the Business Process Execution Language for Web Services (BPEL4WS)[1], a merger of

[1] `http://www-128.ibm.com/developerworks/library/specification/ws-bpel/`

IBM's WSFL and Microsoft's XLANG, as well as ontology-based web-service approaches such as OWL-S[2]. When analyzing the underlying design principles and execution models of business workflow approaches, a focus on control-flow patterns and events becomes apparent, whereas dataflow is often a secondary issue.

Scientific workflow systems, on the other hand, tend to have execution models that are much more dataflow-oriented. Examples include academic systems such as KEPLER [11], Taverna [15], and Triana [12], and commercial systems such as Inforsense's DiscoveryNet, Scitegic's Pipeline-Pilot, and National Instrument's LabView. With respect to their modeling paradigm and workflow execution models, these systems are closer to visual dataflow programming languages for scientific data and services than to the more control-flow and task-oriented business workflow systems, or to their early scientific workflow predecessors [13,1].

The difference between dataflow and control-flow orientation can also be observed in the underlying formalisms. For example, visualizations of business workflows often resemble flowcharts, state transition diagrams, or UML activity diagrams, all of which emphasize events and control-flow over dataflow. Formal analysis of workflows usually involves studying their control-flow patterns [8,5]. Conversely, the underlying execution model of current scientific workflow systems usually resembles dataflow process networks [10], having traditional application areas in digital signal processing and electrical engineering. Dataflow-oriented approaches are applicable at very different levels of granularity, from low-level CPU operations found in processor architectures, over embedded systems, to high-level programming paradigms such as flow-based programming [14]. Scientific workflow systems and visualization pipeline systems can also be seen as dataflow-oriented problem-solving environments [7] that scientists use to analyze and visualize their data.

Actor-Oriented Workflow Modeling in KEPLER. The KEPLER scientific workflow system is an open-source application, with contributing members from various application-oriented research projects. KEPLER aims at developing generic solutions to the process and application-integration challenges of scientific workflows. Figure 1 shows a snapshot of KEPLER running a bioinformatics scientific workflow.

KEPLER extends the PTOLEMY II system, developed for modeling heterogeneous and concurrent systems and engineering applications, to support scientific workflows. In KEPLER, users develop workflows by selecting appropriate components called "actors" (e.g., from actor libraries or by wrapping web services as actors) and placing them on the design canvas, after which they can be "wired" together to form the desired workflow graph. As shown in Figure 1, workflows can also be hierarchically structured. Actors have *input ports* and *output ports* that provide the communication interface to other actors. Control-flow elements such as branching and loops are also supported. A unique feature of PTOLEMY II (and thus of KEPLER) is that the overall execution and component interaction semantics of a workflow is not buried inside the components themselves, but rather factored out into a separate component called a *director*. PTOLEMY II supports a large number of different directors, each one corresponding to

[2] `http://www.daml.org/services/owl-s/`

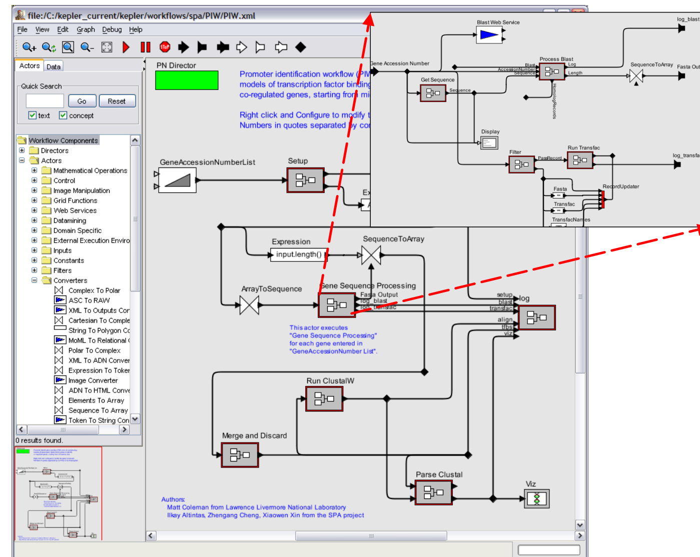

Fig. 1. A bioinformatics workflow in KEPLER: the *composite actor* (center) contains a nested *subworkflow* (upper right); workflow steps include remote service invocation and data transformation; and the execution model is enforced by a *director* (green box)

a unique model of computation. Taken together, workflows, actors, ports, connections, and directors represent the basic building blocks of actor-oriented modeling.

3 A Formal Model of Actor-Oriented Scientific Workflows

This section further defines actor-oriented modeling and its application to scientific workflows. We describe a formal model for scientific workflows and a rich typing system for workflows and workflow components that considers both structural and semantic types. We also briefly describe the use of directors for specifying workflow computation models, which simplifies the task of defining workflows within KEPLER and, along with the typing system, can facilitate the reuse of workflow components.

3.1 Actor-Oriented Hierarchical Workflow Graphs

Workflow Graphs. An actor-oriented *workflow graph* $W = \langle \mathbf{A}, \mathbf{D} \rangle$ consists of a set $\mathbf{A}$ of *actors* representing components or tasks and a set of *dataflow connections* $\mathbf{D}$ connecting actors via data ports. Actors have well defined interfaces and generally speaking, unlike a software agent, are passive entities that given some input data, produce output data (according to their interface). Actors communicate by passing data tokens between their ports.

Ports. Each actor $A \in \mathbf{A}$ has an associated set $\mathsf{ports}(A)$ of *data ports*, where each $p \in \mathsf{ports}(A)$ is either an *input* or *output*, i.e., $\mathsf{ports}(A) = \mathsf{in}(A) \,\dot{\cup}\, \mathsf{out}(A)$ is a disjoint union of *input ports* and *output ports*, respectively. We can think of $\mathsf{ports}(A)$ as the input/output *signature* Σ_A of A, denoted $A :: \mathsf{in}(A) \longrightarrow \mathsf{out}(A)$.[3]

Dataflow Connections. Let $\mathsf{in}(W) = \bigcup_{A \in \mathbf{A}} \mathsf{in}(A)$ be the set of all of input ports of W; the sets $\mathsf{out}(W)$ and $\mathsf{ports}(W)$ are defined similarly. A *dataflow connection* $d \in \mathbf{D}$ is a directed hyperedge $d = \langle \mathbf{o}, \mathbf{i} \rangle$, simultaneously connecting n output ports $\mathbf{o} = \{o_1, \ldots, o_n\} \subseteq \mathsf{out}(W)$ with m input ports $\mathbf{i} = \{i_1, \ldots, i_m\} \subseteq \mathsf{in}(W)$. Intuitively, we can think of $d = \langle \mathbf{o}, \mathbf{i} \rangle$ as consisting of a *merge step* $\mathsf{merge}(d) = \mathbf{o}$ that combines data tokens from the output ports $\mathbf{o}$, and a *distribute step* $\mathsf{distrib}(d) = \mathbf{i}$ that distributes the merged tokens to the input ports $\mathbf{i}$.[4]

A dataflow connection $d = \langle \{o_1\}, \{i_1\} \rangle$ between a single output port and a single input port corresponds to a directed edge $o_1 \xrightarrow{d} i_1$. In general, however, we represent d as an auxiliary connection node having n incoming edges from all output ports $o \in \mathbf{o}$ and m outgoing edges to all input ports $i \in \mathbf{i}$. Dataflow connection $d \in \mathbf{D}$ is called *well-oriented*, if it connects at least one output and one input port. In this way, a directed dataflow dependency between ports is induced.

Workflow Abstraction and Refinement. Abstraction and refinement are crucial modeling primitives. When abstracting a workflow W, we would like to "collapse" it into a single, *composite actor* A_W (hiding W "inside"). Conversely, we might want to refine an actor A by further specifying it via a *subworkflow* W_A, thereby turning A into a composite actor with W_A "inside" (cf. Figures 1 and 3). In both cases, we need to make sure that the i/o-signature Σ_A of the composite actor matches the i/o-signature Σ_W of the contained subworkflow.

Let $W = \langle \mathbf{A}, \mathbf{D} \rangle$ be a workflow. The *free ports* of W are all ports that do not participate in any data connection, i.e., $\mathsf{freeports}(W) := \{p \mid \text{for all } d \in \mathbf{D} : p \notin d\}$. A workflow designer might not want to expose all free ports externally when abstracting W into a composite actor A_W. Instead the i/o-signature is often limited to a subset Σ_W of distinguished ports.

Composite Actors. A *composite actor* A_W is a pair $\langle W, \Sigma_W \rangle$ comprising a *subworkflow* W and a set of distinguished ports $\Sigma_W \subseteq \mathsf{freeports}(W)$, the *i/o-signature* of W. We require that the i/o-signatures of the subworkflow W and of the composite actor A_W containing W match, i.e., $\Sigma_W = \mathsf{ports}(A_W)$.

Hierarchical Workflow Graphs. A *hierarchical workflow* $W = \langle \mathbf{A}, \mathbf{D}, \Sigma \rangle$ is defined like a workflow graph, with the difference that actors might be composite. Inductively, subworkflows can be hierarchical, so that any level of nesting can be modeled. For uniformity, we also include the distinguished i/o-signature Σ of the top-level workflow.

[3] We may also distinguish $\mathsf{par}(A) \subseteq \mathsf{in}(A)$, the *parameter ports* of A, distinct from "regular" data input ports, and used to model different actor "configurations".

[4] The semantics of merging and distributing tokens through dataflow connections is a *separate concern* that is deliberately left unspecified. Instead, this execution semantics is defined separately via *directors*.

3.2 Models of Computation

Following the paradigm of *separation of concerns*, the actor-oriented workflow graphs introduced above only specify communication links (dataflow) between components or tasks (represented by actors), and—in the case of hierarchical workflows—their nesting structure via composite actors. However, the workflow execution semantics or *model of computation* is deliberately left unspecified. In PTOLEMY II a new modeling primitive called a *director* is used to represent the particular choice of model of computation [9].

Thus, we can extend our definition of workflow (graph) W to include a model of computation by means of a director M, i.e., $W = \langle \mathbf{A}, \mathbf{D}, \Sigma, M \rangle$. In the case of the unspecified merge/distribute semantics of a data connection node $d = \langle \mathbf{o}, \mathbf{i} \rangle$ above, a director M may prescribe, e.g., the merge semantics to be one of the following: nondeterministic (the token arrival order is unspecified by M); time-dependent and deterministic (tokens are merged according to their timestamps); or time-independent and deterministic (e.g., "round robin" merging of tokens, or "zipping" together tokens from all input ports, creating a single record token). Similarly, different distribution semantics may be prescribed by M: deterministic copy (replicate each incoming token on all outputs); deterministic round robin (forward a token to alternating outputs); or nondeterministic round robin (randomly choose an output port).

More generally, a model of computation specifies all inter-actor communication behavior, separating the concern of *orchestration* (director) from the concern of *actor execution*. The PTOLEMY II system comes with a number of directors including:

- *Synchronous Dataflow* (SDF): Actors communicate through data connections corresponding to queues and send or receive a fixed number of tokens each time they are fired. Actors are fired according to a predetermined static schedule. Synchronous dataflow models are highly analyzable and have been used to describe hardware and software systems.
- *Process Network* (PN): A generalisation of SDF in which each actor executes as a separate thread or process, and where data connections represent queues of unbounded size. Thus actors can always write to output ports, but may get suspended (blocked) on input ports witout a sufficient number of data tokens. The PN model of computation is closely related to the Kahn/MacQueen semantics of process networks.
- *Continuous Time* (CT): Actors communicate through data connections, which represent the value of a continuous time signal at a particular point in time. At each time point, actors compute their output based on their previous input and the tentative input at the current time, until the system stabilizes. When combined with actors that perform numerical integration with good convergence behavior, such models are conceptually similar to ordinary differential equations and are often used to model physical processes.
- *Discrete Event* (DE): Actors communicate through a queue of events in time. Events are processed in global time order, and in response to an event an actor is permitted to emit events at the present or in the future, but not in the past. Discrete event models are widely used to model asynchronous circuits and instantaneous reactions in physical systems.

3.3 Structural and Semantic Typing of Scientific Workflows

The formal model described above separates the concerns of component communication (*dataflow connections*) from the overall model of computation (a.k.a. *orchestration*), imposed by the director. This separation achieves a form of *behavioral polymorphism* [9], resulting in more reusable actor components and subworkflows. In a sense,

the actor-oriented modeling approach "factors out" the concern of component coordination and centralizes it at the director.

As mentioned in Section 2, scientific workflows are typically data-oriented. The modeling primitives so far, however, have been agnostic about data types. We introduce a novel *hybrid type system* for modeling scientific data that separates *structural data types* and *semantic data types*, but allows them to be explicitly linked using *hybridization constraints*.

Structural Types. Let $\mathcal{S}$ be a language for describing structural data types. For example, $\mathcal{S}$ may be one of XML Schema, XML DTD, PTOLEMY II's token type system, or any other suitable data model or type system for describing structural aspects of data such as the relational model, an object-oriented data model, or a programming language type system (e.g., a polymorphic Hindley-Milner system).

Any port $p \in \mathsf{ports}(W)$ may have a *structural data type* $\mathtt{s} = \mathsf{dt}(p)$, where $\mathtt{s} \in \mathcal{S}$ is a type expression constraining the allowed set of values that the port p can accept (for an input port $p \in \mathsf{in}(W)$) or produce (for an output port $p \in \mathsf{out}(W)$). When using XML Schema as $\mathcal{S}$, e.g., the structural data type of a port is a concrete XML Schema type such as `xsd:date` or any user-defined type. If $\mathcal{S}$ is the relational model, $\mathtt{s}$ describes the tuple or table type of p.

Semantic Types. Let $\mathcal{O}$ be a language for expressing semantic types. By this we mean, in particular, suitable logics for expressing *ontologies*. For example, $\mathcal{O}$ might be a description logic ontology (expressed, e.g., in OWL-DL).

Any port $p \in \mathsf{ports}(W)$ may have a *port semantic type* $\mathrm{C} = \mathsf{st}(p)$, where C denotes a *concept expression* over $\mathcal{O}$. For example, $\mathrm{C}_1 = \mathsf{st}(p_1)$ might be defined as

$$\textsc{Measurement} \sqcap \forall \textsc{itemMeasured}.\textsc{SpeciesOccurrence} \qquad (\mathrm{C}_1)$$

indicating that the port p_1 accepts (or produces) data tokens that are measurements where the measured item is a species occurrence (as opposed to, e.g., a temperature).[5] In addition to port semantic types, any actor $A \in \mathbf{A}$ may also be associated with an *actor semantic type*, categorizing the overall function or purpose of A.[6]

Well-Typed Workflows. Structural and semantic types facilitate the design and implementation of workflows by constraining the possible values and interpretations of data in a scientific workflow W. Another advantage is that the scientific workflow system can validate data connections. For example, if the workflow designer connects two ports $p_1 \xrightarrow{d} p_2$ with structural types $\mathtt{s}_1 = \mathsf{dt}(p_1)$ and $\mathtt{s}_2 = \mathsf{dt}(p_2)$, the system can check whether this connection satisfies the implied subtype constraint $\mathtt{s}_1 \preceq \mathtt{s}_2$. Similarly, for semantic types $\mathrm{C}_1 = \mathsf{st}(p_1)$ and $\mathrm{C}_2 = \mathsf{st}(p_2)$, the system can check whether the implied concept subsumption $\mathrm{C}_1 \sqsubseteq \mathrm{C}_2$ holds.

---

[5] We note that terms within a concept expression may be from distinct ontologies.

[6] Typically the vocabularies chosen for semantic port types and semantic actor types are disjoint, with the former denoting "objects" and the latter denoting "actions" or "tasks".

3.4 Hybrid Types for Scientific Workflows

Structural and semantic types can be considered independently from one another. For example, a workflow designer might start by modeling semantic types and only later in the design process be concerned with structural types (cf. Section 4). Conversely, when reverse-engineering existing executable workflows, structural types might be given first; and only later are semantic types introduced for the purpose of facilitating workflow integration.

Treating semantic and structural types independently offers a number of benefits, and is primarily motivated by the desire to easily interoperate legacy workflow components and components created by independent groups within KEPLER. Decoupling the structural and semantic aspects of workflow types facilitates the use of more standard and generic structural data types, while still allowing the specific semantic constraints of the data to be expressed. Also, one can provide or refine semantic types without altering the underlying structural type, can search for all components having a particular semantic type (regardless of the structural type used), and can provide multiple semantic types for a single component (e.g., drawn from distinct ontologies).

An additional feature of hybrid types is the ability to not only independently consider structural and semantic types, but also interrelate them by a constraint mechanism called *hybridization*. Thus, in general, a hybrid type has three (optional) components, the structural type, the semantic type, and the hybridization constraint.

Formally, let $\mathcal{H}$ be a language of (hybridization) *constraints*, i.e., linking structural and semantic type information. We express constraints from $\mathcal{H}$ in logic, thus requiring that structural and semantic types are expressed in a logic formalism as well. For structural types this means that for any $\mathtt{s} \in \mathcal{S}$ and any logic query expression $e(\bar{x})$ over the set $\mathsf{inst}(s)$ of instances of $\mathtt{s}$, we can evaluate $e(\bar{x})$ on a particular data instance $I \in \mathsf{inst}(s)$, returning a list[7] of variable bindings $[\ \bar{x} \mid I \models e(\bar{x})\ ]$, i.e., those parts of I that satisfy the query $e(\bar{x})$.[8]

For example, given the structural (relational) type $\mathtt{s}_1 = \mathtt{r}(\mathtt{site}, \mathtt{day}, \mathtt{spp}, \mathtt{occ})$ and the above semantic type C_1, the following constraint α_1 "hybridizes" $\mathtt{s}_1$ and C_1:

$$\begin{aligned} &\forall x_{\mathtt{site}}, x_{\mathsf{day}}, x_{\mathsf{spp}}, x_{\mathsf{occ}}\ \exists y:\ \mathtt{r}(x_{\mathtt{site}}, x_{\mathsf{day}}, x_{\mathsf{spp}}, x_{\mathsf{occ}}) \longrightarrow \\ &\qquad \text{MEASUREMENT}(y) \wedge \text{ITEMMEASURED}(y, x_{\mathsf{occ}}) \wedge \\ &\qquad \text{SPECIESOCCURRENCE}(x_{\mathsf{occ}}) \end{aligned} \qquad (\alpha_1)$$

Here, the left-hand side of the implication corresponds to a query expression $e(\bar{x})$ that extracts the item being measured from a relational measurement record. The right-hand side of the implication asserts the existence of a MEASUREMENT y whose ITEMMEASURED x_{occ} is a SPECIESOCCURRENCE. Note that a hybridization constraint such as α_1 can be seen as a "semantic annotation" of the data structure $\mathtt{s}_1$ (the left-hand side of the constraint) with a concept expression (the right-hand side of the constraint).

Exploiting Hybrid Types. By interlinking the otherwise independent structural and semantic type systems, additional inferences can be made. Consider a data connection

[7] We consider variable binding *lists* to accomodate order-sensitive data models such as XML; for unordered models a *set* of bindings can be returned.

[8] Here, $\bar{x} = x_1, \ldots, x_n$ denotes a vector of logical variables.

d that connects two ports $p_1 \xrightarrow{d} p_2$ having *incompatible* structural types $\mathtt{s}_1 = \mathsf{dt}(p_1)$ and $\mathtt{s}_2 = \mathsf{dt}(p_2)$, i.e., where $\mathtt{s}_1$ is *not* a subtype of $\mathtt{s}_2$, denoted $\mathtt{s}_1 \not\preceq \mathtt{s}_2$. Given (hybridization) constraints α_1 and α_2 that map parts of $\mathtt{s}_1$ and $\mathtt{s}_2$ to a common ontology, one can indirectly identify structural correspondences between parts of $\mathtt{s}_1$ and $\mathtt{s}_2$ by "going through the ontology." Technically, this approach is achieved by a resolution-based reasoning technique called the chase.[9]

Exploiting I/O-Constraints. Moreover, for an actor $A \in \mathbf{A}$, a set Φ_{io} of *i/o-constraints* may be given, inter-relating the input and output ports of A. For example, an i/o-constraint can be used to define (or approximate) how values of output ports can be derived from values of input ports. Such a (partial) specification of an actor can be used to propagate hybridization constraints themselves through one or more actors. Assume that $p_1 \in \mathsf{in}(A)$ has the structural type $\mathtt{s}_1 = \mathtt{r}(\mathtt{site}, \mathtt{day}, \mathtt{spp}, \mathtt{occ})$ from above, and $p_2 \in \mathsf{out}(A)$ has a structural type $\mathtt{s}_2 = \mathtt{r}'(\mathtt{sp}, \mathtt{oc})$,[10] and that the following i/o-constraint φ_{io} is given:

$$\forall x_{\mathtt{site}}, x_{\mathtt{day}}, x_{\mathtt{spp}}, x_{\mathtt{occ}} :\ \mathtt{r}(x_{\mathtt{site}}, x_{\mathtt{day}}, x_{\mathtt{spp}}, x_{\mathtt{occ}}) \longrightarrow \mathtt{r}'(x_{\mathtt{spp}}, x_{\mathtt{occ}}) \qquad (\varphi_{io})$$

Using the i/o-constraint φ_{io}, we can now propagate the above constraint α_1 "through" the actor A by applying φ_{io}. We are currently exploring reasoning procedures for propagation that handle a variety of i/o-constraint operations including aggregration, union, and group-by constructs. In this simple example, by applying the propagation procedure, we would obtain a (hybridization) constraint α_2 for the output port p_2 of A:

$$\begin{aligned} &\forall x_{\mathtt{sp}}, x_{\mathtt{oc}}\ \exists y :\ \mathtt{r}'(x_{\mathtt{sp}}, x_{\mathtt{oc}}) \longrightarrow \\ &\qquad \textsc{Measurement}(y) \wedge \textsc{itemMeasured}(y, x_{\mathtt{oc}}) \wedge \\ &\qquad \textsc{SpeciesOccurrence}(x_{\mathtt{oc}}) \end{aligned} \qquad (\alpha_2)$$

Summary. Given the various extensions described above, we can now define a *typed workflow* $W = \langle \mathbf{A}, \mathbf{D}, \Sigma, M, \Phi \rangle$ to also include a set of *constraints* Φ. More precisely, $\Phi = \langle \Phi_{\mathcal{S}}, \Phi_{\mathcal{O}}, \Phi_{\mathcal{H}}, \Phi_{io} \rangle$ consists of a set $\Phi_{\mathcal{S}}$ associating *structural types* from $\mathcal{S}$ to ports in W, $\Phi_{\mathcal{O}}$ associating *semantic types* from an ontology $\mathcal{O}$ to actors and ports, $\Phi_{\mathcal{H}}$ linking structural and semantic types of ports, and finally Φ_{io}, specifying i/o-constraints of actors.

4 Design and Implementation of Scientific Workflows

This section presents a collection of design primitives to support workflow engineering (workflow conceptual design to implementation). Each primitive corresponds to a basic operation over the formal model for actor-oriented scientific workflows. Primitives are described as transformations that return the result of applying an operation to a workflow. Workflow engineers can repeatedly apply these primitives, e.g., via the KEPLER graphical user interface, to create their desired scientific workflow (see Figure 2).

[9] For an early version of our approach, see [4].

[10] The structural types $\mathtt{s}_1$ and $\mathtt{s}_2$ are disconnected (unless an i/o-constraint is given), so one cannot assume the values (or types) of the input match the values (or types) of the ouput.

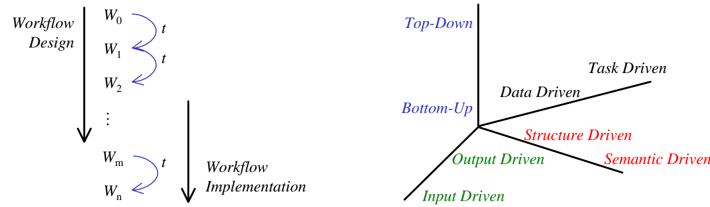

Fig. 2. Workflow engineers evolve workflows by applying design primitives (left), shown as transformations t; and primitives are grouped to form design strategies (right)

Based on the primitives, we identify design strategies to help guide workflow engineers as they develop scientific workflows (see Figure 2). Each strategy emphasizes certain primitives within a larger design process. For example, a particular design method may be divided into a set of phases, and each phase may be guided by a certain strategy.

In this section, we also outline an approach to help automate the implemention of workflow designs. Our approach leverages hybrid typing to refine a workflow into an implemented version by repeatedly applying specific design primitives.

4.1 Scientific Workflow Design Primitives

Basic Actor-Oriented Design Primitives. Figure 3 summarizes the basic actor-oriented modeling primitives. In particular, we include primitives to: introduce new actors and dataflow connections into workflows (transformation t_1); add input and output ports to actors (transformation t_2); refine port structural types (transformation t_3); group (abstract) a portion of a workflow into a composite actor (transformation t_4); define an actor as a composite (transformation t_5); create dataflow connections (transformation t_6); and assign a director to a workflow (transformation t_7). For structural datatype refinement (transformation t_3), we require the "refined" datatype to be a subtype of the existing structural type. Although not shown in Figure 3, we also assume a transformation that "generalizes" structural types (structural type *abstraction*) requiring introduction of appropriate structural supertypes.

Semantic Typing Primitives. Figure 4 summarizes the semantic (hybrid) typing primitives. The first two transformations t_8 and t_9 refine actor semantic types and input and output port semantic types, respectively. Semantic-type refinement requires the introduction of subconcepts, i.e., to refine an actor semantic type T to T', the constraint $\mathrm{T}' \sqsubseteq \mathrm{T}$ must hold. Refining the semantic types of an actor results in specializing the actor's operation. For instance, by refining an input-port semantic type, we further limit the kinds of objects an actor can process. And similarly, by refining an output-port semantic type, we further limit the kinds of objects that can be produced by an actor.

Often, actor and port semantic type refinements are performed together. For example, consider the following series of refinements (each consisting of individual actor and port semantic type refinements):

1. DATAMATRIX → [ANALYSIS] → RESULTSET

Basic Transformations	*Starting Workflow*	*Resulting Workflow*	*Resulting Workflow*
t_1: Entity Introduction (actor or data connection)			
t_2: Port Introduction			
t_3: Datatype Refinement ($s' \preceq s, t' \preceq t$)	s t	s′ t	s t′
t_4: Hierarchical Abstraction			
t_5: Hierarchical Refinement			
t_6: Dataflow Connection			
t_7: Director Introduction			

Fig. 3. Actor-oriented design primitives summarized as transformations where actors are represented as solid boxes; ports as triangles; dataflow connections as circles; composite actors as dashed boxes; and directors as solid (green) boxes

2. PHYLOGENETICMATRIX → [PHYLOGENETICANALYSIS] → PHYLOGENETICTREE
3. NEXUSMATRIX → [CLADISTICANALYSIS] → CONSENSUSTREE

The first refinement states that the semantic type of an actor is ANALYSIS, consisting of an input port of semantic type DATAMATRIX and output port of semantic type RESULTSET. Here, ANALYSIS, DATAMATRIX, and RESULTSET represent general concepts. The second refinement provides more details concerning the actor semantic type, which also influences the input and output port semantic types. The third refinement provides semantic types specific to a particular implementation of an analysis, again influencing the input and output port semantic types.

Primitives t_{10} and t_{11} are used to refine hybridization constraints and i/o-constraints, respectively. Like with semantic types, both hybridization constraint refinement and i/o-constraint strengthening specialize existing hybridization constraints and i/o-constraints (shown as the implications $\alpha' \rightarrow \alpha$ and $\psi \rightarrow \varphi$ in Figure 4).

Similar to the structural type refinement operation, each semantic type refinement operation is assumed to have a corresponding version for abstraction (i.e., generalization of types).

Extended Primitives for Dataflow Connections. It is often convenient to "loosely" connect actors through dataflow connections and then give the details of the connection later as the workflow becomes more complete. The dataflow-connection refinement (transformation t_{12}) provides two approaches for specifying the details of such a connection. The first (shown as the first resulting workflow for the refinment in Figure 4) splits a dataflow-connection node d into two separate dataflow-connection nodes d_1 and d_2 such that:

$$\mathsf{merge}(d_1) \cup \mathsf{merge}(d_2) \equiv \mathsf{merge}(d) \text{ and } \mathsf{distrib}(d_1) \cup \mathsf{distrib}(d_2) \equiv \mathsf{distrib}(d)$$

Extended Transformations	*Starting Workflow*	*Resulting Workflow*	*Resulting Workflow*
t_8: Actor Semantic Type Refinement ($T' \sqsubseteq T$)	*T*	*T¢*	
t_9: Port Semantic Type Refinement (C′ ⊑ C, D′ ⊑ D)	C D	C′ D	C D′
t_{10}: Annotation Constraint Refinement ($a' \rightarrow a$)	a_1 C D a_2 s t	a¢ C D a_2 s t	a_1 C D a¢ s t
t_{11}: I/O Constraint Strengthening ($y \rightarrow j$)	*j*	*y*	
t_{12}: Dataflow Connection Refinement			
t_{13}: Adapter Insertion			
t_{14}: Actor Replacement	*f*	*f¢*	
t_{15}: Workflow Combination (Map)	f_1 f_2	f_1 f_2	

Fig. 4. Additional primitives to support scientific-workflow design and implementation, where adapters are shown as solid, rounded boxes

The second refinement transforms a dataflow-connection node d into an actor node A, which is constructed from d as follows: (1) each port p in $\mathsf{merge}(d)$ generates a new port p' that is added to $\mathsf{in}(A)$; (2) a new dataflow-connection node is created to connect the ports p and p'; (3) a new port p'' is created and added to $\mathsf{out}(A)$; and (4) $\mathsf{merge}(d)$ is assigned the singleton set $\{p''\}$.

Although not shown in Figure 4, we assume both versions of dataflow-connection refinement have corresponding generalization primitives.

Primitives for Adapter Insertion. The adapter insertion primitive (transformation t_{13}) is used to insert special actors called *adapters* between incompatible dataflow connections. We focus on adapters for situations in which a connection contains a semantic or structural incompatibility.

A *semantic adapter* is used to align input and output port connections that do not satisfy the subconcept typing constraint. We consider two cases for semantic adapter insertion. In the first case, an output port with semantic type C is connected to an input port with semantic type D. We assume that C and D are incompatible such that the constraint C ⊑ D does not hold. For example, let C and D be defined as follows.

$$\mathrm{C} \equiv \textsc{Measurement} \sqcap \forall \textsc{itemMeasured}.\textsc{SpeciesOccurrence}$$
$$\mathrm{D} \equiv \textsc{Measurement} \sqcap \forall \textsc{itemMeasured}.\textsc{SpeciesRichness}$$

The first actor produces data containing species' occurrence measurements and the second actor consumes data containing species' richness measurements. The semantic types are not compatible because SPECIESOCCURRENCE is not a subconcept of

SPECIESRICHNESS. In general, however, richness data can be obtained from occurrence data through a simple conversion, namely, by summing occurrrence.

In this case, one may choose to insert a semantic adapter between the two actors. Conceptually, the adapter provides a data conversion that can reconcile the semantic differences between the two actors. Typically the input and output semantic types of a semantic adapter will be assigned the corresponding actor output and input, respectively. A semantic adapter can also have a more general input semantic type (e.g., a semantic type $\mathrm{C}' \sqsupseteq \mathrm{C}$) and a more restrictive output semantic type (e.g., $\mathrm{D}' \sqsubseteq \mathrm{D}$).

A *structural adapter* is similar to a semantic adapter, but is used to reconcile incompatible structural types found in data connections (as opposed to incompatible semantic types). Within KEPLER, users can determine whether connections are created that are semantically or structurally incompatible. Incompatible types can be fixed by: (1) inserting an appropriate adapter; (2) modifying the data connection; or (3) abstracting and/or refining the problem types.

Primitives for Actor Replacement. The actor replacement primitive (transition t_{14}) is used to "swap" one actor in a workflow with another actor. We use standard object-oriented inheritance rules [6] to determine when a particular actor replacement is appropriate. Figure 5 shows three simple cases: the general case of safe replacement (shown on the left), unsafe replacement (shown in the middle), and context-sensitive replacement (shown on the right). For general replacement, an actor A_1 can be replaced by another actor A_2 if the following conditions hold: [11]

1. A_2 has an input (output) port for each of A_1's input (output) ports[12];
2. A_2's actor semantic type is a subconcept of A_1's actor semantic type;
3. A_2's input port types are equivalent or more general than A_1's; and
4. A_2's output port types are equivalent or more specific than A_1's.

As shown in Figure 5, *unsafe replacement* occurs when the semantic (or structural) port types do not satisfy the above conditions. However, unsafe replacement may still be considered appropriate when the replacement is taken in context. That is, the general form of unsafe replacement (the middle case of Figure 5) may become safe when the surrounding data connections are considered. We call this case *context-sensitive replacement*, as shown in Figure 5, the input and output semantic (and structural) replacement rules are determined by the semantic (and structural) types of corresponding data connections.

Primitives for Combining Workflows. The workflow combination primitive (transformation t_{15}) is used to assemble two or more workflows into a single "conglomerate." To be combined, the input and output structural and semantic types of the separate workflows must be combatible. The most specific input types of the separate workflows are used as the combined-workflow input types; and the most general output types of the

[11] Note that in general we also require the i/o-constraint f' of the replacement to imply the i/o-constraint f of the original actor (i.e., $f' \rightarrow f$).

[12] Here, A_2 may contain more output ports than A_1, and possibly more input ports so long as the "extra" ports are not required. As future work, we are also more generally considering matching aggregrations of ports.

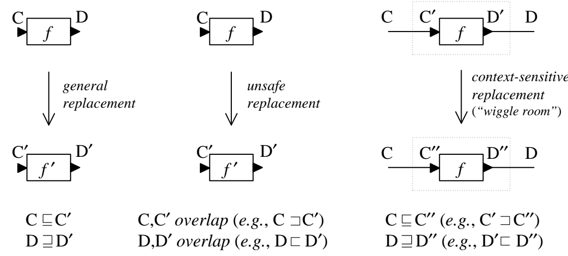

Fig. 5. Semantic type constraints for general, unsafe, and context-sensitive replacement

separate workflows are used as the combined-workflow output types. Combining similar workflows is useful for cases where multiple algorithms exist to perform a similar function, e.g., to perform multiple multivariate statistics over the same input data.

The workflow combination primitive is similar to the higher order function `map :: [a] -> (a -> b) -> [b]`, which returns the result of applying a function to each element of a list. In particular, the workflow combination primitive can be viewed as a variant `Map :: a -> [(a -> b)] -> [b]` that takes a value v and a list of functions $f_1, f_2, ..., f_n$, and returns a list containing the values $f_1(v), f_2(v), ..., f_n(v)$.

4.2 Strategies for Workflow Design

As shown in Figure 2 (and similar in spirit to [3]), we define high-level design strategies that emphasize specific transformation primitives. The strategies can be used to describe design methods where at each stage, a particular strategy (a point in the design space of Figure 2) is applied. The design strategies are defined as follows.

- *Task-Driven Design*: Workflow engineers focus on identifying the conceptual actors of a workflow. This strategy can involve defining actor ports, semantic types, structural types, associations, and i/o-constraints along with hierarchcial refinements and replacements to convert abstract actors to implemented versions.
- *Data-Driven Design*: Wofkflow engineers focus on identifying the input data and dataflow connections of workflows. Dataflow connections may be elaborated using refinement.
- *Semantic-Driven Design*: Workflow engineers focus on specifying the semantic types of the workflow. The engineer may start with a “blank” workflow topology containing basic actors and dataflow connections, and identify the appropriate semantic types, adding concepts and roles to ontologies as needed.
- *Structure-Driven Design*: Like semantic-driven design, but for structural types.
- *Input-Driven Design*: Workflow engineers focus on identifying the input of a workflow, and design from “left to right,” i.e., from the input side to the output side of the workflow.
- *Output-Driven Design*: Like input-driven design, but focus on data products first.
- *Top-Down Design*: Workflow engineers focus on refining actors and dataflow connections. The engineer may begin with a single empty workflow and iteratively apply hierarchical and dataflow connection refinement.

– *Bottom-Up Design*: Workflow engineers focus on abstraction of actors and dataflow connections. The engineer may first define specific parts of a workflow and iteratively abstract the workflow using hierarchical abstraction to connect the various parts.

Different workflow design methods apply in different situations. We have found that the process of re-engineering existing applications into workflows often starts with top-down, structure driven strategies. But, when scientists develop new workflows (e.g., new analyses as opposed to "re-engineered" ones), a mix of semantic, input, and output strategies are used.

4.3 From Design to Implementation of Scientific Workflows

Here we outline an approach that leverages hybrid typing, replacement rules, and adapter insertion to help automate the task of finding appropriate actor implementations for workflow specifications. We assume there is a repository $\mathbf{R}$ of semantically typed actors and workflows. We use the term *abstract actor* to refer to actors that cannot be executed (i.e., without implementations) and *concrete actor* to refer to executable actors. $\mathbf{R}$ may consist of abstract or concrete actors, composite actors, and entire workflows. The following steps sketch the approach for finding implementations of a workflow W:

1. if W is a concrete workflow, output W
2. select an abstract actor $A_T \in \mathbf{A}$ that has an actor replacement $A_C \in \mathbf{R}$
3. let W' be the workflow that results from replacing A_T by A_C
4. if W' has an incompatible dataflow connection, insert an abstract adapter
5. repeat with $W := W'$

The basic idea of the approach is to define a search space such that each node represents a workflow and transitions between nodes are defined using steps 2-4 above. The procedure for finding implementations of W is to navigate the search space (e.g., using a breadth-first or depth-first search algorithm) looking for nodes that represent concrete workflows. In the transitions (steps 2-4) defined above, we replace individual abstract actors in a workflow with valid replacements from the respository. When a concrete actor is inserted that violates a semantic or structural typing rule, we also insert an abstract adapter actor, which can also be replaced (in subsequent steps). In general, for a given worfklow W there may be many associated concrete workflows, depending on whenever an abstract actor can be replaced by more than one repository element. The user may wish to combine some or all of the resulting workflows using the workflow combination primitive.

5 Summary

This paper extends our previous work by describing a formal model of scientific workflows based on actor-oriented modeling and design. The approach facilitates conceptual modeling of scientific workflows through a novel hybrid type system, and by providing a set of primitive modeling operations for end-to-end scientific workflow development. Our approach can also support the conceptual and structural validation of scientific workflows, as well as the discovery of type-conforming workflow implementations

via replacement rules and by inserting appropriate semantic and structural adapters for workflow integration. Much of this work is currently implemented within the KEPLER system, and we are currently extending KEPLER with semantic propagation and additional reasoning techniques to further exploit hybrid types.

References

1. A. Ailamaki, Y. E. Ioannidis, and M. Livny. Scientific Workflow Management by Database Management. In *Proc. of SSDBM*, pages 190–199, 1998.
2. G. Alonso and C. Mohan. Workflow Management Systems: The Next Generation of Distributed Processing Tools. In *Advanced Transaction Models and Architectures*. 1997.
3. C. Batini, S. Ceri, and S. Navathe. *Conceptual Database Design: An Entity-Relationship Approach*. Benjamin/Cummings, 1992.
4. S. Bowers and B. Ludäscher. An Ontology-Driven Framework for Data Transformation in Scientific Workflows. In *Proc. of the Intl. Workshop on Data Integration in the Life Sciences (DILS)*, volume 2994 of *LNCS*, pages 1–16. Springer, 2004.
5. F. Casati, S. Ceri, B. Pernici, and G. Pozzi. Conceptual Modelling of WorkFlows. In *Object-Oriented and Entity-Relationship Modelling Conference (OOER)*, volume 1021 of *LNCS*, pages 341–354. Springer, 1995.
6. G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3), 1995.
7. K. W. B. H. Wright and M. J. Brown. The Dataflow Visualization Pipeline as a Problem Solving Environment. In *Virtual Environments and Scientific Visualization*. Springer, 1996.
8. B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. Ph.D. Thesis, Queensland University of Technology, 2002.
9. E. A. Lee and S. Neuendorffer. Actor-oriented Models for Codesign: Balancing Re-Use and Performance. In *Formal Methods and Models for Systems*. Kluwer, 2004.
10. E. A. Lee and T. M. Parks. Dataflow process networks. *Proc. of the IEEE*, 83(5):773–801, 1995.
11. B. Ludäscher, I. Altintas, D. H. Chad Berkley, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice and Experience, Special Issue on Scientific Workflows*, 2005. to appear.
12. S. Majithia, M. S. Shields, I. J. Taylor, and I. Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. In *Proc. of the IEEE Intl. Conf. on Web Services (ICWS)*. IEEE Computer Society, 2004.
13. J. Meidanis, G. Vossen, and M. Weske. Using Workflow Management in DNA Sequencing. In *Proc. of CoopIS*, pages 114–123, 1996.
14. J. P. Morrison. *Flow-Based Programming: A New Approach to Application Development*. Van Nostrand Reinhold, 1994.
15. T. M. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, R. M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.
16. W. van der Aalst and K. van Hee. *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*. MIT Press, 2002.
17. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.
18. M. zur Muehlen. *Workflow-based Process Controlling: Foundation, Design, and Application of workflow-driven Process Information Systems*. Logos Verlag, Berlin, 2004.