CORE
Metadata, citation and similar papers at core.ac.uk
Provided by The University of Utah: J. Willard Marriott Digital Library

VisMashup: Streamlining the Creation of Custom Visualization Applications

Emanuele Santos, Lauro Lins, James P. Ahrens, Juliana Freire, and Cláudio T. Silva, Senior Member, IEEE

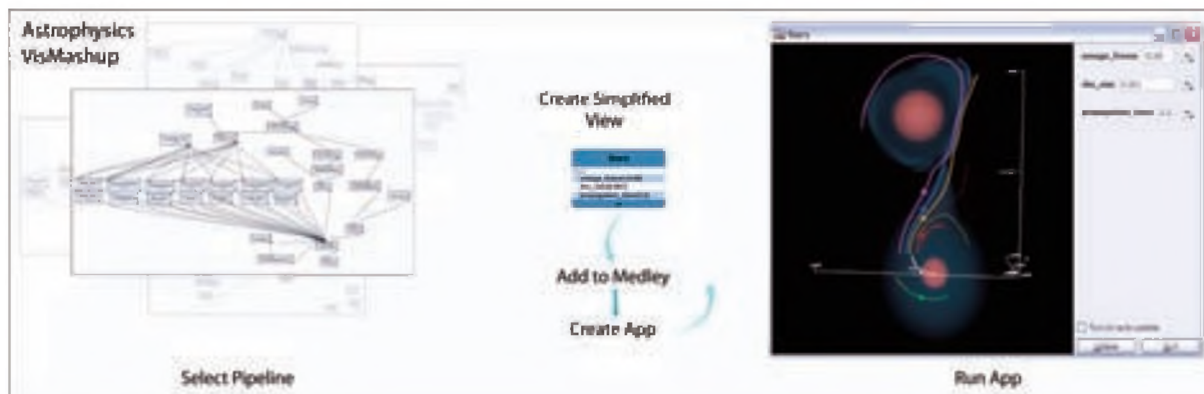

Fig. 1: Creating an Astrophysics VisMashup to explore a binary star system.

Abstract—Visualization is essential for understanding the increasing volumes of digital data. However, the process required to create insightful visualizations is involved and time consuming. Although several visualization tools are available, including tools with sophisticated visual interfaces, they are out of reach for users who have little or no knowledge of visualization techniques and/or who do not have programming expertise. In this paper, we propose VISMASHUP, a new framework for streamlining the creation of customized visualization applications. Because these applications can be customized for very specific tasks, they can hide much of the complexity in a visualization specification and make it easier for users to explore visualizations by manipulating a small set of parameters. We describe the framework and how it supports the various tasks a designer needs to carry out to develop an application, from mining and exploring a set of visualization specifications (pipelines), to the creation of simplified views of the pipelines, and the automatic generation of the application and its interface. We also describe the implementation of the system and demonstrate its use in two real application scenarios.

Index Terms—Scientific Visualization, Dataflow, Visualization Systems.

1 INTRODUCTION

Data exploration through visualization is an effective means to understand and obtain insights from large collections of data. Not surprisingly, visualization has become a mature area with an established research agenda [21], and several systems have been developed that support the creation of complex visualizations [6,12,14,20,22,32,35]. But even using systems that have sophisticated visual programming interfaces, such as AVS, DX, SCIRun, and VisTrails, the path from the raw data to insightful visualizations is laborious and error-prone. And this has hampered a wider adoption of visualization techniques.

Visual programming interfaces expose computational components as *modules* and allow the creation of complex visualization pipelines which combine these modules in a dataflow, where *connections* between modules express the flow of data through the pipeline [16]. They also provide built-in constraint checking mechanisms (e.g., that disallow a connection between incompatible module ports) that simplify the creation of pipelines. Recently, new techniques have been proposed to simplify the creation and refinement of these pipelines, including recommendation systems [15,27] and a mechanism to refine pipelines by analogy [24]. Notwithstanding, without detailed knowledge of the underlying computational components, it is difficult to understand what series of modules and connections ought to be added to obtain a desired result. Consequently, the generation of visualizations is a task that is best carried out by experts who are familiar with visualization techniques and tools.

Recognizing this problem, there have been a number of efforts to create simpler visualization applications (e.g., [8,28,32]) that are customized to a particular task (e.g., [5,13,18]) and that use concepts that are more meaningful to domain experts than visualization primitives. Some systems [8,18,28,32] provide customized interfaces that allow users to control the parameters of a pipeline. These simplified interfaces are easier to use than the general systems and can be designed specifically for domain experts that do not have expertise in visualization. Nonetheless, they are expensive to create and maintain. The lifecycle for developing custom applications is often long, requiring several iterations until the application does what the end-user needs. Whereas such solutions can be (and have been) successfully applied in big scientific projects, they are out of reach for smaller-scale efforts. The costs involved in creating these applications make them unsuitable for one-of-a-kind, exploratory visualizations.

Our Approach. In this paper we propose VISMASHUP, a new approach for simplifying the creation, maintenance, and use of customized visualization applications (or mashups). Because VISMASHUP uses dataflows as the underlying model, in principle, it can be combined with visualization systems that use dataflows to

• *Emanuele Santos, Lauro Lins, Juliana Freire, and Cláudio T. Silva are with the Scientific Computing and Imaging (SCI) Institute at the University of Utah. email: {emanuele, lauro, juliana, csilva}@sci.utah.edu.*
• *James P. Ahrens is with Los Alamos National Lab in Los Alamos, NM. email: ahrens@lanl.gov.*

Manuscript received 31 March 2009; accepted 27 July 2009; posted online 11 October 2009; mailed on 5 October 2009.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org .

Authorized licensed use limited to: The University of Utah. Downloaded on February 23,2010 at 14:52:16 EST from IEEE Xplore. Restrictions apply.

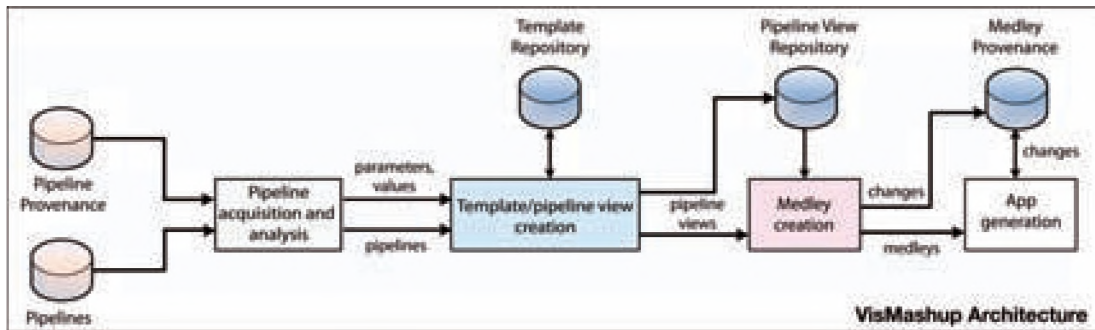

Fig. 2: The VisMashup Architecture.

specify pipelines (e.g., VTK-based systems such as ParaView [14] and VisTrails [35]). In addition to pipelines defined as dataflows, VISMASHUP requires the visualization systems to provide access to pipeline specifications, the ability identify and change pipeline components, and to execute the pipelines. As shown in Figure 2, the VISMASHUP system consists of a set of components that support tasks an application designer has to perform to construct a custom application. The *Pipeline acquisition and analysis* component allows designers to query and mine pipeline collections and their provenance [9]. This enables them to identify relevant pipelines, discover interesting parameters that should be exposed in the application, as well as a set of valid values that can be used for these parameters. Through a point-and-click interface, the *Template/pipeline view creation* component allows a designer to manipulate pipelines and create simplified views which expose only the pipeline components that are relevant for a given task. Pipeline views can then be combined into a medley (see Sec. 3.2) to create more complex, exploratory tasks (*Medley creation*). The *App generation* component uses the medley specification to automatically generate the application and associated user interface. VISMASHUP also maintains detailed provenance information of the application development process and use (*Medley Provenance*). As we discuss in Section 3, this information can be used to further refine and improve the derived apps.

While this infrastructure simplifies the job of an application designer, it gives an end-user the flexibility to create visualizations through a simple interface that hides unnecessary details of the underlying pipelines and how they are integrated. Furthermore, end-users also have the ability to customize a mashup, by hiding and showing components as well as moving them around in the interface.

Outline. The remainder of the paper is organized as follows. We review related work in Sec. 2. In Sec. 3, we introduce the basic concepts and describe the main components of the VISMASHUP model. The implementation details of the VISMASHUP components and their interfaces are presented in Sec. 4. In Sec. 5, we present two case studies where we applied the VISMASHUP system: a mashup that allows users to interact with astrophysics analyses; and a mashup that allows neuroscientists to explore and compare complex visualizations for a study on the effects of repetitive Trans-cranial Magnetic Stimulation (rTMS) on working memory. We conclude in Sec. 6, where we outline directions for future work.

2 RELATED WORK

A number of visualization systems have been proposed that provide sophisticated visual programming interfaces. These interfaces aim to simplify the creation of visualizations by allowing users to create pipelines which combine computational components (modules) in a dataflow [12, 22, 32, 35]. However, without detailed knowledge of the underlying computational components, it is difficult to understand what series of modules and connections ought to be added to obtain a desired result. Other systems, such as ParaView [14], go a step further and not only hide the details of the pipelines, but also provide abstractions that simplify the creation of visualizations. However, these systems are still very general and require detailed knowledge of visualization techniques. Similar to these tools, VISMASHUP relies on dataflows which describe a series of visualizations, and like ParaView, the derived applications hide the underlying pipelines. However, with VISMASHUP, customized (specific) applications can be derived that are targeted to end-users who do not necessarily have knowledge of visualization.

Another approach that has been used to simplify data exploration through visualization is the creation of custom applications. CDAT, for example, is a tool designed for climatologists that supports visual exploration of climate data [5]. Because such applications are costly to develop, this approach is not scalable. Instead of building applications from scratch, some systems (e.g., AVS [32], IRIS Explorer [8], VISSION [28] and SCIRun [18]) allow the creation of customized interfaces for visualization pipelines. In SCIRun, these interfaces are called PowerApps. BioFEM, for example, is a PowerApp built atop of a SCIRun pipeline that solves a bioelectric forward problem using the finite element method provided by Biological Problem Solving Environment (BioPSE) tools [18]. BioFEM consolidates all necessary user controls in a single UI, hiding other generic controls and the associated dataflow network. Unfortunately, a SCIRun PowerApp needs to be manually crafted for a given pipeline. IRIS Explorer provides a GUI to facilitate the layout and design of widgets representing parameters, but users need to manually create the interfaces. Similar to VISSION, VISMASHUP automatically derives the application interface from a dataflow network. But VISMASHUP goes a step further: it provides a comprehensive infrastructure that allows the application developer to explore collections of pipelines and combine them to create the applications. Another important distinction between our approach and previous systems is that it captures detailed provenance of both application development and use.

Several approaches have been proposed whose goal is to simplify application development, from high-level programming languages such as Ruby[1] and Python[2], to visual programming tools such as Visual Basic. These, however, still require programming expertise. More recently, there have been a number of proposals for tools targeted to non-programmers. For example, Yang et al. [37] proposed a WYSIWYG tool that aids in the development of data-driven Web applications; and Yahoo! Pipes [36] provides a visual interface through which users can assemble pipelines for constructing Web mashups. Although VISMASHUP shares some of the same goals, our approach targets the construction of applications which use dataflows as basic building blocks.

Reduced-functionality interfaces, such as training wheels [4], have been used to help first-time and novice users of complex systems. In these interfaces, advanced features are disabled on the GUI to prevent user errors and confusion. Shneiderman [25] proposed the design of layered interfaces, which provide a set of increasingly complex interfaces, allowing the user to select the interface that best suits her needs. VISMASHUP naturally supports both techniques—the designer can easily create several variations of an application through a series of pipeline views. In addition, because provenance is maintained, detailed information about how the different application versions were created and their differences is readily available.

[1] http://www.ruby-lang.org
[2] http://www.python.org

Authorized licensed use limited to: The University of Utah. Downloaded on February 23,2010 at 14:52:16 EST from IEEE Xplore. Restrictions apply.

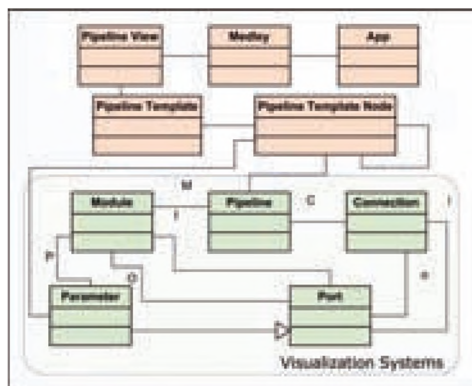

Fig. 3: Conceptual schema of the main concepts in VisMashup.

3 Model

In what follows, we present the basic concepts underlying the model we propose for generating customized visualization apps.

Dataflows. We assume that the dataflow [16] model is used to specify visualization pipelines. In this model, a pipeline is represented as a directed acyclic graph (DAG) where nodes (modules) represent computations and arcs (connections) denote data dependencies. If there is an arc from node m to node n, n requires the output of m to execute. Besides the outputs of earlier computations, nodes can also have parameters (data values) as inputs. The dataflow model is widely-used in visualization systems, including ConMan [10], AVS [32], SCIRun [22], and VTK-based systems such as Paraview [14], VisIt [34] and VisTrails [35]. Basing VisMashup on dataflows enables our framework to be combined with virtually any system that uses dataflows to specify pipelines. A schematic representation of the model is depicted in Figure 3, and we describe its components below.

Definition 1. A *pipeline* (or dataflow) d is defined by a tuple (M, C), where M is a set of *modules* and C a set of *connections*. Each module $m \in M$, in turn, is associated with a tuple (I_m, O_m, P_m), where I_m corresponds to a set of *input ports*, O_m corresponds to a set of *output ports*, and P_m is a list of *parameters*. Each parameter $p \in P_m$ is associated with a value v. In a *connection* $(o, i) \in C$ connecting module m to module n, $o \in O_m$ and $i \in I_n$, o is called the *source port* and i is called the *target port*. *Sources* are modules that do not have target ports and *sinks* are modules not containing source ports. Additionally, ports and parameters have a type; and if there is a connection $(o, i) \in C$, the types of o and i must be compatible. ■

Definition 2. A *subpipeline* $d_s(M_s, C_s)$ of a pipeline $d(M, C)$ is the pipeline induced by the set of modules $M_s \subset M$. For each connection $c \in C$, if c connects two modules m_1 and m_2, and $m_1, m_2 \in M_s$, then $c \in C_s$. ■

3.1 Pipeline Operations

The two basic operations our model applies to pipelines are: *run*, which executes a pipeline; and *substitution*, which manipulates pipeline components (i.e., parameter values and modules). These operations are used both during application design and to manipulate the pipeline through the application.

Run. A *pipeline run* is the execution of a pipeline in the order determined by the network of modules and connections, in a *demand-driven* fashion. Each sink module ensures that its input ports have up-to-date data from incoming connections, causing the modules of the source ports of those connections to do the same. These data requests are propagated up (recursively) to the sources of the pipeline which will in turn execute and produce the output values required by the downstream modules. The downstream modules are executed, in turn, until the sinks located at the end of the pipeline are reached.

Substitution. *Substitution* allows parameter values and modules to be replaced within a pipeline. Let $d = (M, C)$ and $d' = (M', C')$ be two pipelines. The operation $substituteParameter_d(p, v)$ assigns value v to parameter p in pipeline d, provided that the types of v and p are compatible.

The operation $substitutePipeline_d(M_s, d', f)$ replaces the subpipeline induced by the modules in $M_s \subset M$ by the pipeline d', using the mapping function f to rebuild the connections from d' to M. Let $C_s \subset C$ be the set of connections which connect modules in M to modules in M_s. Given a connection $(o_s, i_s) \in C_s$, where o_s is an output port of a module m_{source} *in* M, and $m_{source} \notin M_s$, i_s is replaced by the appropriate port in d', $f(i_s)$, provided the types are compatible. For connections $(o_s, i_s) \in C_s$, where i_s is an input port of a module m_{source} *in* M, and $m_{source} \notin M_s$, o_s is replaced by the appropriate port in d', $f(o_s)$, provided the types are compatible.

3.2 Templates, Views, Medleys and VisMashups

Visualization pipelines can be very complex, making it difficult for users other than the original developer to modify them. Consider for example, the relatively simple pipeline shown in Figure 4(a). This pipeline has 6 modules and tens of parameters. To explore the visualizations that can be derived by this pipeline, the user must be able to identify the relevant parameters that relate to a particular visualization feature, know the ranges of values that can be used for individual parameters as well as valid combinations of values for different parameters. This can be very challenging, in particular for users that do not have detailed knowledge of visualization techniques and of the individual VTK modules. An important goal of VisMashup is to simplify this task and provide the end-user the flexibility to explore different visualizations in a more intuitive way. Below, we introduce concepts and operations that allow a pipeline designer to create simplified views of pipelines and pipeline collections (medleys) which abstract away details that are not important for a given task. Using these operations, the designer may also select and expose configurable components of the pipelines which will guide the end user in providing valid inputs as well as experiment with different pipeline variations.

Pipeline templates. We introduce the notion of a *pipeline template* as an extension of a pipeline that allows users to define reconfigurable pieces of the pipeline in a hierarchical way. Users can select and label parameters or subpipelines using a nomenclature that is meaningful for a given application or task. Figure 4(b) shows a pipeline template generated for the VTK pipeline in Figure 4(a). In order to give the end user the ability to modify the colors and rotations, the designer selected to expose the colors in the `vtkProperty` module with the parameter names `color_R`, `color_G` and `color_B`; and the X and Y rotations in `vtkActor` as `rotateX` and `rotateY`, respectively. In addition, selecting the subpipeline containing the module `vtkCylinderSource` allows this subpipeline to be replaced by other pipelines through the customized application.

The root of the template hierarchy represents the pipeline, and its children and descendants correspond to configurable parameters and subpipelines. We refer to each element in the template hierarchy as a *pipeline template node*. Nodes that correspond to subpipelines are represented as rectangles and parameters as ellipses. Note that labels are unique in a given hierarchy level. By representing the template as a hierarchy, our approach is able to handle arbitrary nesting of pipelines.

Pipeline template nodes provide the same operations of a pipeline as well as other specific operations for labeling and removing labels, for creating, adding and removing child nodes, creating and removing connections between template nodes and between template nodes and modules, and for materializing a pipeline.

Pipeline Views. Whereas in a pipeline template important and configurable elements (i.e., parameters and subpipelines) are selected, in a *pipeline view*, a new (abstract) module is created that hides all details of the underlying pipelines, including modules, connections and parameters that are deemed as unnecessary for deriving a set of visualizations required by an end user.[3] As we describe below, we use *medleys* as a mechanism to synchronize pipeline parameters and combine multiple views. More formally, a *pipeline view* d_v is a projection of a pipeline d, where only a subset of the pipeline elements (modules, parameters, and subpipelines) are exposed for direct interaction. For example, a user cannot change a parameter if that parameter is not

[3] Pipeline views in VisMashup are similar to macro pipelines in AVS [32].

Authorized licensed use limited to: The University of Utah. Downloaded on February 23,2010 at 14:52:16 EST from IEEE Xplore. Restrictions apply.

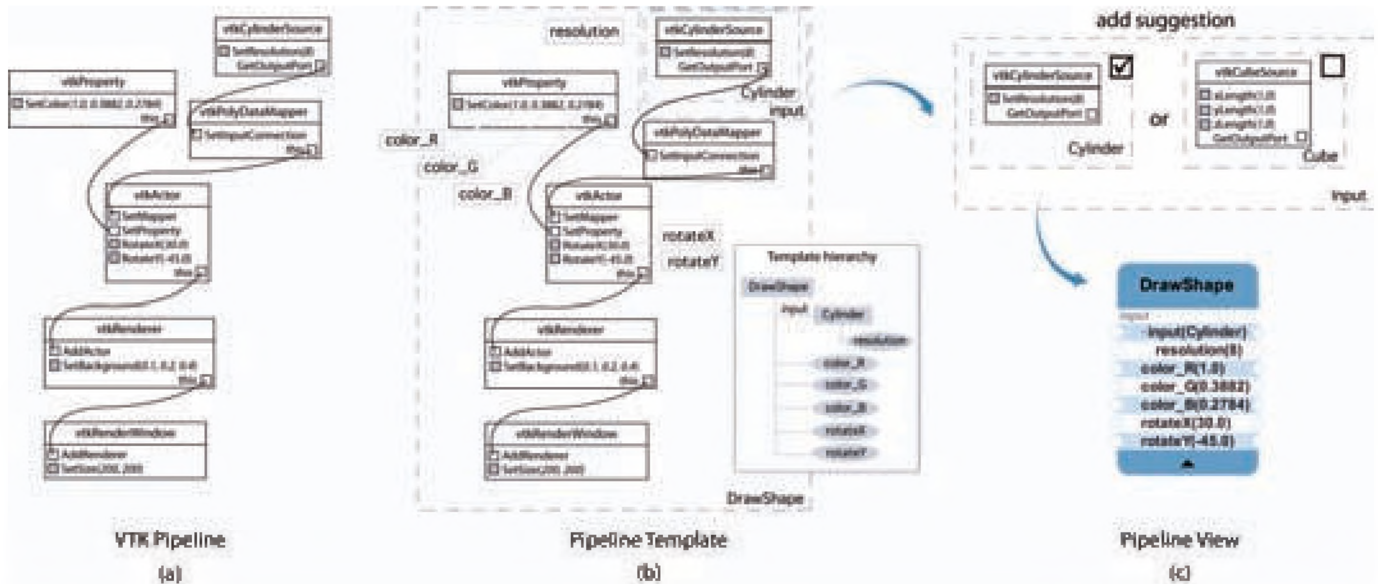

Fig. 4: Example of how pipeline views are created. (a) Original VTK pipeline that draws a simple cylinder. (b) The user marks configurable pieces of the pipeline to create a template. (c) A pipeline view is created based on the pipeline template. The user defines template node *Cube* as another suggestion for variable input and *Cylinder* as the default value.

exposed by the pipeline view. However, a pipeline view still keeps a reference to the original pipeline so the operations applied to the view can also be applied to the pipeline, including execution. We refer to the exposed elements as *variables*.

Notice that a pipeline view can be naturally built on top of a pipeline template. In fact, the view parameters and configurable subpipelines are also represented as a hierarchy that mirrors the one for the template hierarchy. Besides setting the visibility of template nodes and parameters, a developer can also specify a list of *suggestions* for the values a variable can take—these suggestions are used to derive application widgets that enable users to conveniently select a value from list and *bind* it to the variable (see Sec. 4). Figure 4(c) shows a view (`DrawShape`) derived from the template in Figure 4(b). In addition, the view also contains suggestions for alternative subpipelines that can be used to bind the `input` variable, i.e., `vtkCylinderSource` and `vtkCubeSource`. When defining these suggestions for template nodes, the developer also specifies how these suggestions connect in the parent template. It is possible that a given template node can be bound in different ways and the presence of some variables in a pipeline view is determined by the bindings used for their parents. In the example shown in Figure 4(c), the developer defined different types of source, each containing their own set of parameters. There are two possible bindings suggested for the variable `input`: the subpipelines `Cylinder` and `Cube`. In the pipeline view shown, where `Cylinder` is used, its parameter (i.e., `resolution`) is added to the view. If `Cube` were chosen, its parameters would be included instead.

Medleys. For exploratory visualization tasks, a user often needs to create and manipulate a set of pipelines. For example, to compare different isosurface extraction algorithms, several pipelines need to be created, including one pipeline per isosurface algorithm, and a pipeline that produces a histogram which accumulates the quality information for each mesh. Figure 5 illustrates two visualizations and their corresponding quality histograms.

In order to support the construction of applications that require the combination of multiple pipelines, we introduce the notion of a medley: a *medley* **M** is a collection of (related) pipeline views. VisMashup supports a set of operations for manipulating the views in a medley. One such operation is *synchronization*. A variable x in a pipeline view $d_v \in \mathbf{M}$ can be synchronized with any variable x' in another view $d'_v \in \mathbf{M}$ if x and x' have the same type. If x and x' are synchronized, binding either to a value v sets both variables to v.

The ability to synchronize variables is useful, in particular, for tasks

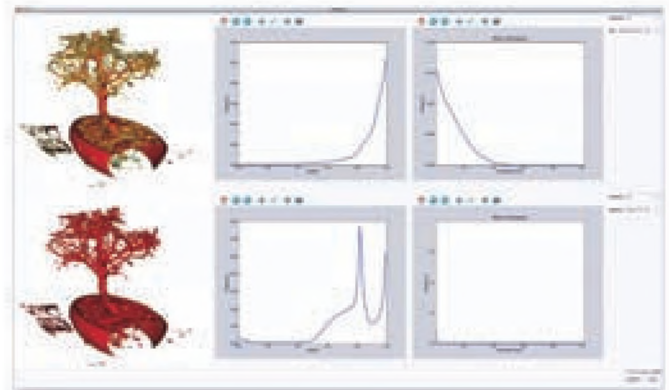

Fig. 5: Comparing visualizations derived by two different isosurface algorithms and their associated quality histograms.

such as comparative visualization, since it helps ensure that all parameters across different pipelines whose values should be the same are indeed the same. Consider, again, the isosurface example. Given a medley that contains the views for the pipelines that use the different isosurface algorithms, by synchronizing their input file and contour values, in a single step, the user could set these parameters and they would be automatically propagated to the two pipelines. Furthermore, synchronization enables a user to efficiently try out different configurations. Instead of setting values for each pipeline individually—which can be both time consuming and error prone, the value for a parameter is set only once and it is automatically propagated to all synchronized variables in multiple views. Another possible operation that can be applied to a medley is *composition*. Composition can be achieved by synchronizing an output port of a view to the input port of another. In our example, composition could be used to pass the mesh derived by the two isosurface views to the view that derives the quality histograms. Note that because pipeline views reduce the number of components that are exposed for modification, they make it easier to identify how pipelines can be integrated and synchronized.

VisMashups. A *vismashup* (application) is a flexible GUI automatically generated from a medley specification. Instead of interacting directly with a dataflow network or a very general and complex GUI, users manipulate and execute a set of pipelines in a medley through a small number of graphical widgets. Figures 8 and 9 show examples

Authorized licensed use limited to: The University of Utah. Downloaded on February 23,2010 at 14:52:16 EST from IEEE Xplore. Restrictions apply.

Fig. 6: Window to control the mining options and filtering conditions to select pipelines.

of vismashups. Application maintenance is simplified, since when the underlying medley changes, the GUI can be automatically updated. Furthermore, besides setting parameter values and synchronizing parameters, users can customize a mashup: they can hide, show, and move widgets around. Changes made to the pipeline views through the mashup are propagated down to the pipeline level and sent to the visualization system for execution.

An important component of the mashup generation and execution sub-systems is that they maintain detailed provenance information. VISMASHUP maintains provenance of the mashup design process—it keeps all versions and variations of a mashup created as the visualization expert interacts with a user, using a model similar to the change-based provenance proposed in [19]. Besides, log information of the use of a mashup is also kept (e.g., when the mashup was executed, the values provided). The log information can be used to further refine the views and medleys.

4 THE VISMASHUP SYSTEM

We have built a prototype VISMASHUP system using Python, Qt, PyQt, and VisTrails as the underlying dataflow system. VisTrails supports a flexible set of libraries, including VTK, ITK, matplotlib, SciPy and Web services. As illustrated in Figure 2, the VISMASHUP system has four main components: pipeline acquisition and analysis; template and view creation; medley creation; and application generation. The implementation of these components and their user interfaces are described in detail below.

4.1 Pipeline Acquisition and Analysis

Although the pipelines for a mashup can be built from scratch, often, as visualization experts (designers) interact with domain experts (users), they construct a series of pipelines to explore different aspects of a data set, or different ways to visualize the data. The actual pipelines and their provenance, including information about the different runs, parameters set and their values, can be useful for designing the views that serve as the basis for the custom application. The Pipeline Acquisition and Analysis component allows the designer to explore a pipeline collection to identify relevant pipelines as well as identify important parameters and associated values.

Based on the metadata and provenance associated with a pipeline, we provide a filtering mechanism for developers to select pipelines satisfying simple conditions (see Figure 6). These conditions can be based on *time* (when the pipeline was created), the *user* that created the pipeline, *execution* (whether a pipeline was successfully executed and when), *tags* and *notes* associated with a pipeline. These rules are compiled into a simple regular expression matching over the text descriptions of the pipelines. Note that it is also possible to integrate more advanced interfaces that allow structural queries over the pipelines (e.g., [2, 24]). Besides filtering, this component also provides an algorithm to help organize the pipeline collection into groups and extract information about parameter changes (see below). If there is provenance information about parameters that have been set across different executions of a pipeline and their values, this information can be explored both to suggest which parameters to expose in the pipeline view as well as to create a range of values that can be used as suggestions for the exposed parameters (Sec. 3.2).

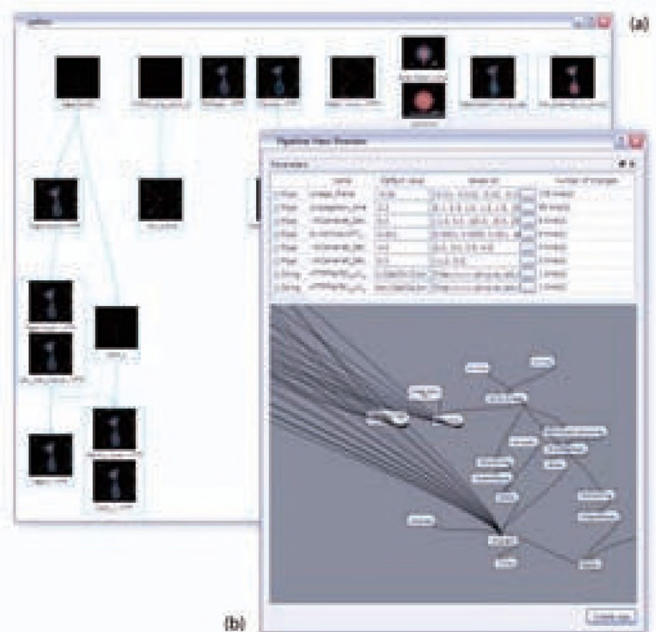

Fig. 7: VisMashup Mining Interface: (a) Relevant pipelines organized in a Hasse Diagram; (b) Examining the parameters returned from the mining process using the Template/Pipeline View Editor.

Given a set of pipelines D, the procedure GROUP-AND-MINE-PARAMETERS initially groups together pipelines that have the same structure (modules and connections), i.e., isomorphic graphs (line 2). For each group, the parameters and values stored in the pipelines and in their provenance (if available) are extracted and attached to the group (lines 3-12). Finally a *Hasse diagram* [3, 31] of the groups is generated by calling procedure BUILD-HASSE-GRAPH. The Hasse diagram is a tree where each node corresponds to a group of isomorphic graphs, and edges between a parent and a child node indicate containment. It shows how pipelines are related with respect to structure: the graph of a parent node is contained in the graph of the child node. Figure 7(a) shows an example of a Hasse diagram for a collection of 21 pipelines. Each pipeline is identified by a label and a thumbnail of the visualization it generates. Although the Hasse diagram resembles an image graph [17], the latter only captures the relationship between visualization parameter changes and are created for a single pipeline.

```
GROUP-AND-MINE-PARAMETERS(D)
 1  hasse-nodes ← MAKE-NODES-FROM-PIPELINES(D)
 2  hasse-nodes ← MERGE-WITH-SAME-STRUCTURE(hasse-nodes)
 3  for n in hasse-nodes
 4    n.param-list ← DICTIONARY()
 5    for d in n.pipelines
 6      params ← GET-PARAMS-FROM-PROVENANCE(d)
 7      params ← params + d.parameters
 8      for p in params
 9        if p.name not in n.param-list.KEYS()
10          n.param-list[p.name] = [ ]
11        n.param-list[p.name] ← p.value)
12  BUILD-HASSE-GRAPH(hasse-nodes)
```

To discover the containment relationships, it is necessary to compute subgraph isomorphism between all pipeline group pairs. For our prototype, we implemented the algorithm proposed by Ullman [31]. We note that, although computing subgraph isomorphism is computationally hard for general graphs,[4] we have observed that, in practice, this computation can be efficient for pipelines represented as dataflows. Just to give an idea, for a collection of over 7,000 pipelines, it took 140 seconds to generate their Hasse diagram. Besides, even if

[4]This problem is trivially reducible from the MAX-CLIQUE problem, a well-known NP-complete problem [11].

Authorized licensed use limited to: The University of Utah. Downloaded on February 23,2010 at 14:52:16 EST from IEEE Xplore. Restrictions apply.

this computation takes long, it can be done in an offline process once for a collection, and be re-used multiple times.

The interface for the Hasse diagram is interactive, with zoom and pan functionality. This diagram gives a developer an overview of the collection and shows which pipelines share the same structure but have different parameter values. By clicking on a node of the diagram, the developer has access to another screen that provides details about all the pipelines in that group. The pipeline is displayed together with all the parameters that were changed, the number of times they were changed, and the set of values used to bind them. If the pipeline stores information about its executions, a plot of the execution times is also shown. This information is then used to help the construction of pipeline templates and views (Figure 7(b)).

4.2 Template and Pipeline View Creation

After selecting the interesting pipelines and extracting information about important parameters and values, the developer can use the *Pipeline Template Editor* to define the configurable pieces of the pipelines, by labeling subpipelines and parameters. To select a pipeline to be used as a template, the designer simply clicks on the corresponding node in the Hasse diagram. This causes the Template Editor to be invoked. When the developer labels a parameter or a subpipeline, all the relevant information associated with these components that had been collected during the mining process is automatically added to the template (e.g., all the values associate with the parameter). This is illustrated in Figure 7(b), which shows a set of parameters extracted from one of the pipelines in Figure 7(a).

Once a template is created, one of the operations supported by the Template Editor is view creation. While configuring a view, users can set the visibility of the parameters and configurable subpipelines, as well as select suggestions from the list stored in the template. Note that the information about both values as well as change frequency is very useful for configuring a view. Clearly, parameters that have had many different values are good candidates for being exposed. As shown in Figure 7(b), the value for parameter `omega_frame` was modified 178 times, while the parameter `propagation_time` changed 98 times. The values for these parameters can be used to create a list of suggestions for variables in the view. As we describe below, suggestions are included in the mashup to guide the user interaction, e.g., besides typing a value in the interface, the user can also select a value from a list of suggestions. Note that both templates and views are stored in a repository where they can be accessed at a later time.

4.3 Medley Creation and Mashup Generation

To combine pipeline views in a medley, the designer uses the *Medleys Editor*. The views stored in the Pipeline View Repository are displayed on a panel and they can be dragged and dropped on a canvas. Once on the canvas, the medley operations (synchronization and composition) can be applied to the views. A screenshot of the Medleys Editor is shown in Figure 8.

Each variable in a view has an associated handle—the circles on the left and right of each variable name in the pipeline view in Figure 4(c). By connecting the handles for two variables in distinct views, their values are synchronized. To simplify the task of identifying variables to be synchronized, when the designer starts a connection all the variables that are compatible with that variable are highlighted.

VISMASHUP uses the change-based provenance model [19] to capture all the changes made to a medley. We designed and implemented a change algebra that captures the actions applied to a medley (e.g., synchronize variables) as well as to its components (e.g., hide/expose view variables).

A mashup is generated from a medley using the following algorithms:

GENERATE-APP(*medley*)

1 *widgets* = TREE()
2 *window* = CREATE-WINDOW(*medley*)
3 **for** *pipeline-view* **in** *medley.pipeline-views*
4 BUILD-GUI-WIDGETS(*pipeline-view.root, window, widgets*)
5 **for** *sync* **in** *medley.syncs*
6 LINK-WIDGETS-FROM-SYNC(*sync*)

BUILD-GUI-WIDGETS(*template-node, parent, node*)

1 *widget* = CREATE-CONTAINER(*parent, template-node*)
2 *child* = MAKE-TREE-NODE(*widget*)
3 *node*.ADD-CHILD(*child*)
4 **for** *var* **in** *template-node.children*
5 BUILD-GUI-WIDGETS(*var, widget, child*)
6 **for** *par* **in** *var.parameters*
7 *widget-type* ← GET-FROM-REGISTRY(*par.type*)
8 *wp* ← CREATE-WIDGET(*widget-type, widget, par*)

GENERATE-APP automatically derives a user interface for a given medley. The interface consists of a set of widgets represented as a tree whose root corresponds to the mashup window. For each view in the medley, the algorithm iterates through the view components and derives their widgets by invoking BUILD-GUI-WIDGETS (lines 3-4). After the widgets are created, the widgets that correspond to synchronized variables in the medley are linked (lines 5-6) —this ensures that when the value is set for a widget w, the same value is propagated to all widgets linked to w.

BUILD-GUI-WIDGETS generates the widgets for a pipeline view. It first creates a container for the view (line 1), and then recursively builds a widget tree following the view template tree (Sec. 3.2). The widget types associated to the types of the variables are stored in a *Registry*. A widget is selected for a variable based on the mapping stored in the Registry. In the current implementation, the Registry is populated based on the types available in VisTrails, where designers can define their custom types and widgets. A planned extension to VISMASHUP is to allow the designer to edit the Registry and configure the choices independently of the underlying dataflow system. In addition to the widgets, canvases are created to display each of the outputs of a view. Note that the algorithms above are used to generate both client applications as well as Web-based mashups. The main differences between the two kinds of apps is that the communication between a Web app and a medley is done via a Web service and it provides a more limited interaction with the visualizations.

4.4 Mashup Interaction

When a mashup is created, a default layout is selected for the widgets. By activating the mashup edit mode, the user can move the widgets around, hide and show widgets, enable and disable synchronizations. The new custom layout is saved as an attribute of the medley associated with the mashup, and the next time the mashup is invoked, the custom layout is loaded.

When a container widget changes its value (e.g., changing from Cylinder to Cube in the pipeline view described in Figure 4), the mashup window needs to be updated accordingly. As the widgets in the window follow the hierarchy of the pipeline template, updates can be applied locally to that node—it is not necessary to update the other widgets in the window. The update procedure is shown below.

UPDATE-APP(*window, tree − node*)

1 *parent* ← *tree-node.parent*
2 *template-node* ← *tree-node.widget.template-node*
3 BUILD-GUI-WIDGETS(*template-node, parent.widget, parent*)
4 REMOVE-LINKS-FROM-WIDGETS()
5 **for** *sync* **in** *window.medley.syncs*
6 LINK-WIDGETS-FROM-SYNC(*sync*)

Note that before UPDATE-APP is called, the medley associated with that window detects the change events and stores the corresponding provenance information. Since variables can be removed, the synchronization information may need to be updated as do the associated links between widgets. Finally, executing a mashup causes all pipeline views to update their corresponding pipelines using the values on the medley (the same as in the GUI) and trigger their execution.

5 Case Studies

5.1 Sharing Astrophysics Analyses

Professor Joel Tohline's group at Louisiana State University (LSU) has been developing computational fluid dynamics (CFD) techniques

Authorized licensed use limited to: The University of Utah. Downloaded on February 23,2010 at 14:52:16 EST from IEEE Xplore. Restrictions apply.

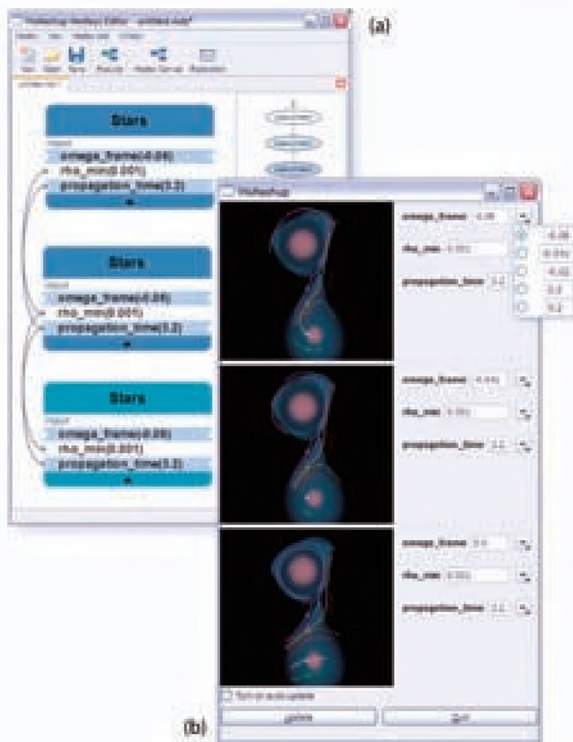

Fig. 8: Astrophysics VisMashup for comparing different visualizations. (a) Medley created for the VisMashup using 3 instances of a pipeline view with parameters ρ_{min} and *propagation time* synchronized. (b) Using the VisMashup interface, the user explores different values for Ω_{frame}. From top to bottom, the values used are -0.06, -0.041, and 0.0.

to model various astrophysical phenomena, including simulation of mass-transfer instabilities in binary star systems [7]. They use scientific visualization tools and techniques to help them explore the results of these complex CFD simulations. They use VisTrails to construct complex pipelines that involve VTK modules to generate streamlines within each binary mass-transfer simulation, and custom analysis modules for identifying values for important parameters associated with the flows [29, 30].

A binary star system consists of two stars orbiting around a common center of mass with an orbital period P. When the system is viewed from a frame that is rotating with an angular frequency $\Omega_{frame} = 2\pi/P$, the system will appear to be stationary. During a simulation, as mass and angular momentum are transferred from one star to the other, the binary system's orbital period together with the value of Ω_{frame} is expected to vary. Using the most recent pipelines they developed [29], the scientists are able to examine the properties of binary mass-transfer flows from reference frames having a range of different angular frequencies $\Omega_{frame} = (\Omega_0 + \Delta\Omega)$, where Ω_0 is the frame of reference's angular frequency at the beginning of the simulation. As a result, they are able to determine which value of $\Delta\Omega$ provides the best measure of the true orbital period of the binary star system.

The LSU group has published these pipelines on their Web site [29], where they encourage people to try the different values of Ω_{frame} and examine the behavior of the streamlines. Together with the pipelines, they provide annotations where they describe the relevant parameters and values. But to try these pipelines, users must download and install VisTrails, and also learn to use the system. In contrast, using VisMashup, a simple mashup can be created based on the LSU pipelines and deployed as a Web application. Users can then conveniently try the different Ω_{frame} values using simple form-based interface in a Web browser.

The first mashup we developed was based on the pipeline shown in Figure 1. A limitation of this mashup is that, since it contains a single 3D view, it does not easily support the comparison of different parameters. To address this problem, we added more views of this same pipeline to the medley and synchronized the parameters ρ_{min} and *propagation time* in all of them (see Figure 8(a)). The new mashup is more suitable for the exploration of the Ω_{frame} parameter, allowing multiple visualizations to be displayed and compared side by side. The app, shown in Figure 8(b), has three 3D views with a set of widgets for manipulating the parameters of each view. Users can then set a different value for Ω_{frame} in each window and compare the streamlines. This comparison is facilitated by the synchronization of the 3D view cameras.

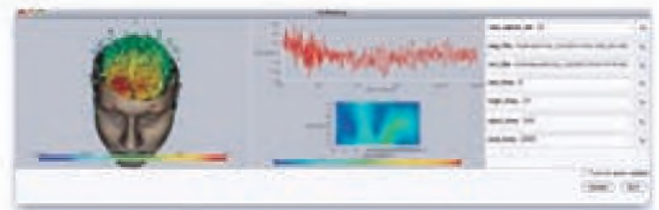

Fig. 9: Neuroscience VisMashup. This mashup combines two pipelines—one that produces a plot of a single EEG sensor's raw data and its S-Transformed representation, and another that creates a volume rendered visualization. To ensure that the plots and the visualization are derived for the same patient, the variables in the pipeline views corresponding to the patient (i.e., the input data set) are synchronized.

5.2 Visualization in Neuroscience

The study of cognition and memory in humans has been an area of intense study in recent years. Current literature shows that a wide variety of psychological disorders have a marked decrease in working memory performance and that a primary indicator of working memory is the spectral properties of the alpha band (8-12 Hz) in the dorsal lateral pre-frontal cortex (DLPFC) [26]. One method under development to help improve working memory performance is repetitive Trans-cranial Magnetic Stimulation (rTMS). rTMS has shown to be capable of positively manipulating the underlying spectral dynamics of human alpha rhythms to influence working memory performance [1].

In order to adequately measure the effects of rTMS on individual alpha-band spectral dynamics and explore signal patterns in the desired region of the brain, Electro-encephalography (EEG) and Magneto-encephalography (MEG) are employed in conjunction with Magnetic Resonance Imaging (MRI). Commonly in collaborative studies, the data processing and visualization experts are unable to adequately analyze the results and implications of the analysis pipelines they create. Likewise, the scientists who can analyze the spectral properties of cognitive dynamics do not have the experience required to write the robust pipelines needed in the study.

VisMashup supports collaboration among the different experts. Experts in data processing and visualization generate a series of pipeline views which expose *only* the functionality the scientists need. The scientists analyzing the results can then explore these views without having to deal with complexities of the corresponding pipelines.

Figure 9 shows a mashup constructed from a medley that consists of two pipelines: one that generates a plot of a single EEG sensor's raw data and its S-Transformed representation, and another that creates a volume rendered image of a patient's brain. By using vismashups like this, the psychiatrist conducting this study was able to explore the patients' data more freely and more extensively without having to learn how to use the underlying visualization system. The visualization experts working on this study also benefited from using VisMashup because it allowed them to generate the final apps more quickly, requiring less time spent training the scientist to manipulate complex pipelines.

Before having access to VisMashup, the expert responsible for creating the multi-modal visualizations needed in this study had designed had a single pipeline consisting of several disjoint sub-pipelines. Each of these sub-pipelines handled a single data modality or processing technique. But for medleys to be effective, it is im-

Authorized licensed use limited to: The University of Utah. Downloaded on February 23,2010 at 14:52:16 EST from IEEE Xplore. Restrictions apply.

portant that the pipelines are modularized. Since medleys are composed by distinct pipeline views, a medley consisting of composite pipelines is inherently difficult to manipulate. By enforcing stricter modularization on the development of pipelines, medley-enabled systems can benefit not only from more concise pipeline descriptions, but from more flexible display of the results.

6 CONCLUSION

We have described VISMASHUP, a new framework that simplifies the creation of custom visualization applications. Using VISMASHUP, an application developer can quickly assemble custom applications by leveraging an existing collection of visualization pipelines and their provenance. The mashups produced by VISMASHUP are not a substitute for more comprehensive, domain-specific applications such as CDAT. Instead, our target are simpler, exploratory applications.

Although VISMASHUP simplifies the development of one-of-a-kind, domain-specific visualization applications, it has some limitations. The integration of different libraries can sometimes be complicated by a number of practical issues, such as compatibility of underlying interface toolkit (e.g., Tk vs Qt). Also, while we try to make sensible choices on the automatic layout of the interface, and we allow the users to fine tune it, this does not guarantee that the most appropriate and intuitive interface for the task at hand is created.

There are several areas of future work that we would like to pursue. First and foremost, we need to perform a more detailed evaluation of the effectiveness and usability of VISMASHUP. To do so, we will carry out user studies. To provide more flexibility in filtering and selecting relevant pipelines, we plan to integrate more sophisticated interfaces that allows structural queries to be specified by example [2, 24]. We also plan to explore alternative techniques for mining the pipeline collection. For example, it can be useful to support different grouping/clustering strategies [23]. Last, but not least, we would like to investigate environments where users can share and collaboratively develop applications. Such environments have been successfully deployed for Web mashups (see e.g., http://www.programmableweb.com). But we believe that the ability to share visualization applications, in addition to data, will help advance existing efforts on collaborative data analysis [33] and it also has the potential to broaden the use of visualization techniques.

ACKNOWLEDGMENTS

We thank Joel Tohline for the astrophysics, and Erik Anderson and Gil Preston for the neuroscience case study. We thank the VisTrails team members for their help with this work. Our research has been funded by the National Science Foundation (grants IIS-0905385, IIS-0746500, ATM-0835821, IIS-0844546, CNS-0751152, IIS-0713637, OCE-0424602, IIS-0534628, CNS-0514485, IIS-0513692, CNS-0524096, CCF-0401498, OISE-0405402, CCF-0528201, CNS-0551724), the Department of Energy SciDAC (VACET and SDM centers), and IBM Faculty Awards (2005, 2006, 2007, and 2008). E. Santos is partially supported by a CAPES/Fulbright fellowship.

REFERENCES

[1] E. Anderson, G. Preston, and C. Silva. Towards development of a circuit based treatment for impaired memory: A multidisciplinary approach. *IEEE EMBS Conf on Neural Engineering*, pages 302–305, 2007.

[2] C. Beeri, A. Eyal, S. Kamenkovich, and T. Milo. Querying business processes. In *VLDB*, pages 343–354, 2006.

[3] G. Birkhoff. *Lattice theory*. American mathematical society, Providence, RI, revised edition, 1948.

[4] J. M. Carroll and C. Carrithers. Training wheels in a user interface. *Communications of the ACM*, 27(8):800–806, 1984.

[5] Climate Data Analysis Tools (CDAT). http://www-pcmdi.llnl.gov/software-portal/cdat.

[6] H. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. Miller, B. J. Whitlock, and N. Max. A contract-based system for large data visualization. In *IEEE Visualization*, pages 190–198, 2005.

[7] M. D'Souza, P. Motl, J. Tohline, and J. Frank. Numerical Simulations of the Onset and Stability of Dynamical Mass transfer in Binaries. *The Astrophysical Journal*, 643(1):381–401, 2006.

[8] D. Foulser. IRIS Explorer: A framework for investigation. *ACM SIGGRAPH Computer Graphics*, 29(2):13–16, 1995.

[9] J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for computational tasks: A survey. *Computing in Science & Engineering*, 10(3):11–21, May-June 2008.

[10] P. E. Haeberli. ConMan: A Visual Programming Language for Interactive Graphics. In *Proceedings of SIGGRAPH '88*, pages 103–111, 1988.

[11] J. Hastad. Clique is hard to approximate within $n^{1-\varepsilon}$. *Acta Mathematica*, 182:105–142, 1999.

[12] IBM. OpenDX. http://www.research.ibm.com/dx.

[13] C. Johnson, R. MacLeod, S. Parker, and D. Weinstein. Biomedical computing and visualization software environments. *Communications of the ACM*, 47(11):64–71, 2004.

[14] Kitware. The Visualization Toolkit (VTK) and Paraview. http://www.kitware.com.

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

[16] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, 1995.

[17] K.-L. Ma. Image graphs- a novel interface for visual data exploration. In *IEEE Visualization*, pages 81–88, 1999.

[18] R. Macleod, D. Weinstein, J. de St. Germain, D. Brooks, C. Johnson, and S. Parker. SCIRun/BioPSE: Integrated problem solving environment for bioelectric field problems and visualization. In *Proceedings of the Int. Symp. on Biomed. Imag.*, pages 640–643, 2004.

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

[20] Mercury Computer Systems. Amira. http://www.amiravis.com.

[21] T. Munzner, C. Johnson, R. Moorhead, H. Pfister, P. Rheingans, and T. S. Yoo. NIH-NSF visualization research challenges report summary. *IEEE Computer Graphics and Applications*, 26(2):20–24, 2006.

[22] S. G. Parker and C. R. Johnson. SCIRun: a scientific programming environment for computational steering. In *Supercomputing*, 1995.

[23] E. Santos, L. Lins, J. P. Ahrens, J. Freire, and C. T. Silva. A first study on clustering collections of workflow graphs. In *IPAW*, pages 160–173, 2008.

[24] C. Scheidegger, D. Koop, H. Vo, J. Freire, and C. Silva. Querying and creating visualizations by analogy. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1560–1567, 2007.

[25] B. Shneiderman. Promoting universal usability with multi-layer interface design. In *Proceedings of the Conf on Universal Usability*, pages 1–8, 2003.

[26] H. Tan, W. Choo, S. Fones, and W. Chee. Neuropsychological performance in schizotypal personality disorder: Importance of working memory. *The American Journal of Psychiatry*, 162(10):1896–1903, 2005.

[27] A. Telea and J. van Wijk. SMARTLINK: An agent for supporting dataflow application construction. In *Proceedings of IEEE VisSym 2000*, pages 189–198, 2000.

[28] R. Telea and J. Van Wijk. VISSION: An Object Oriented Dataflow System for Simulation and Visualization. *Proceedings of IEEE VisSym 1999*, pages 95–104, 1999.

[29] J. Tohline. LSU using VisTrails. http://www.phys.lsu.edu/ tohline/vistrails/.

[30] J. Tohline, J. Ge, W. Even, and E. Anderson. A customized python module for CFD flow analysis within VisTrails. *Computing in Science & Engineering*, 11(3):68–73, 2009.

[31] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[32] C. Upson, J. Thomas Faulhaber, D. Kamins, D. H. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, 9(4):30–42, 1989.

[33] F. B. Viegas, M. Wattenberg, F. van Ham, J. Kriss, and M. McKeon. ManyEyes: A site for visualization at internet scale. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1121–1128, 2007.

[34] VisIt Visualization Tool. https://wci.llnl.gov/codes/visit.

[35] The VisTrails Project. http://www.vistrails.org.

[36] Yahoo! Pipes. http://pipes.yahoo.com.

[37] F. Yang, N. Gupta, C. Botev, E. F. Churchill, G. Levchenko, and J. Shanmugasundaram. Wysiwyg development of data driven web applications. *Proceedings of the VLDB Endowment.*, 1(1):163–175, 2008.

Authorized licensed use limited to: The University of Utah. Downloaded on February 23,2010 at 14:52:16 EST from IEEE Xplore. Restrictions apply.