Scientific Mashups: Runtime-Configurable Data Product Ensembles

Bill Howe[1]*, Harrison Green-Fishback[2], and David Maier[2]

[1] University of Washington (billhowe@cs.washington.edu)
[2] Portland State University ({hgmf, maier}@cs.pdx.edu)

Abstract. Mashups are gaining popularity as a rapid-development, reuse-oriented programming model to replace monolithic, bottom-up application development. This programming style is attractive for the "long tail" of scientific data management applications, characterized by exploding data volumes, increasing requirements for data sharing and collaboration, but limited software engineering budgets.

We observe that scientists already routinely construct a primitive, static form of mashup—an ensemble of related visualizations that convey a specific scientific message encoded as, e.g., a Powerpoint slide. Inspired by their ubiquity, we adopt these conventional data-product ensembles as a core model, endow them with interactivity, publish them online, and allow them to be repurposed at runtime by non-programmers.

We observe that these scientific mashups must accommodate a wider audience than commerce-oriented and entertainment-oriented mashups. Collaborators, students (K12 through graduate), the public, and policy makers are all potential consumers, but each group has a different level of domain sophistication. We explore techniques for adapting one mashup for different audiences by attaching additional context, assigning defaults, and re-skinning component products.

Existing mashup frameworks (and scientific workflow systems) emphasize an expressive "boxes-and-arrows" abstraction suitable for engineering individual products but overlook requirements for organizing products into synchronized ensembles or repurposing them for different audiences. In this paper, we articulate these requirements for scientific mashups, describe an architecture for composing mashups as interactive, reconfigurable, web-based, visualization-oriented data product ensembles, and report on an initial implementation in use at an Ocean Observatory.

1 Introduction

A *mashup* is a web-based, lightweight, situational application integrating services and data that were not necessarily designed to interoperate. The term was coined to describe web applications developed by the general public exercising Google's Map API, but has come to refer to any "quick and dirty" web application based on pre-existing data or services.

The popularity of mashups is attributable to the enormous rate of collective data acquisition. Gray and Szalay argued that derived data dominates the total

* Work performed while author was at Oregon Health & Science University.

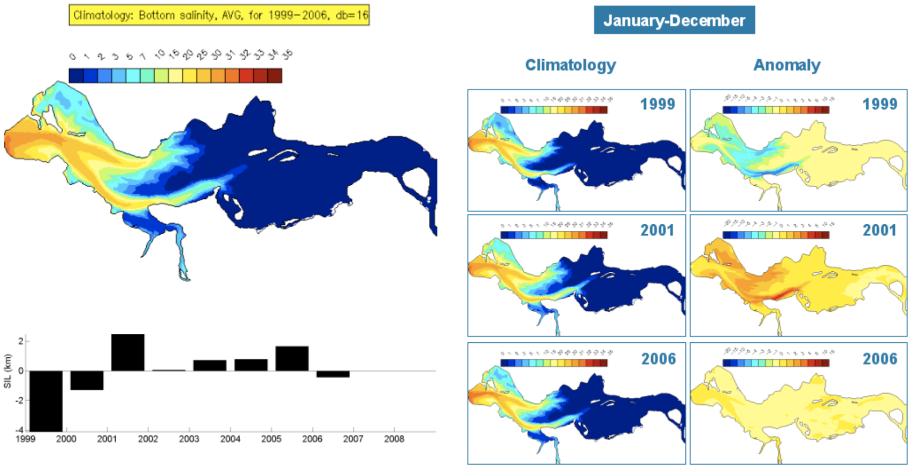


Fig. 1. A data product ensemble from a physical oceanography presentation on "climatologies" (long-term averages). Each map displays the Columbia River Estuary with the Pacific Ocean cropped at left and colored by bottom salinity. The large map at left is the bottom salinity averaged over the entire time period 1999-2006. The six maps at right are arranged in two colums: the left column is the yearly average, and the right column the average bottom salinity subtracted from the overall average to give an indication of variability from the mean.

data volume due to pairwise comparisons [4]. That is, N source datasets leads to $O(N^2)$ derived comparison datasets. Similarly, the number of mashups deployed on the web scales as $O(N^2)$ in the number of services available. For example, one application overlays crime scenes on Google's satellite images (via a join on location) [2], another links your Twitter microblog with your Flickr photostream (via a join on time) [15], another integrates your bank statements with your investment portfolio [11], and so on. The quadratic growth rate in the number of applications must be balanced by a reduction in development time, recruitment of a new class of developer, a higher degree of reuse, or all three.

Of course, personal data management and social networking applications are relatively simple problems — the amount of data processed in each operation is small (a single RSS message, a screenful of search results), and the analysis to be performed is predictable (items equipped with a latitude and longitude are displayed on a map, images are displayed in the browser using thumbnails). Many mashup frameworks rely crucially on these simplifying assumptions [6, 12, 20], making them inappropriate for *enterprise mashups* characterized by larger datasets, specialized users, and domain-specific processing [7]. For example, an enterprise mashup might be required access data from relational databases, spreadsheets, documents, and other in-house proprietary data sources in order to display the combined data in a business-specific format whereas a "consumer grade" mashup will usually access a smaller set of standard data formats, but produce result designed to be as generally accessible as possible.

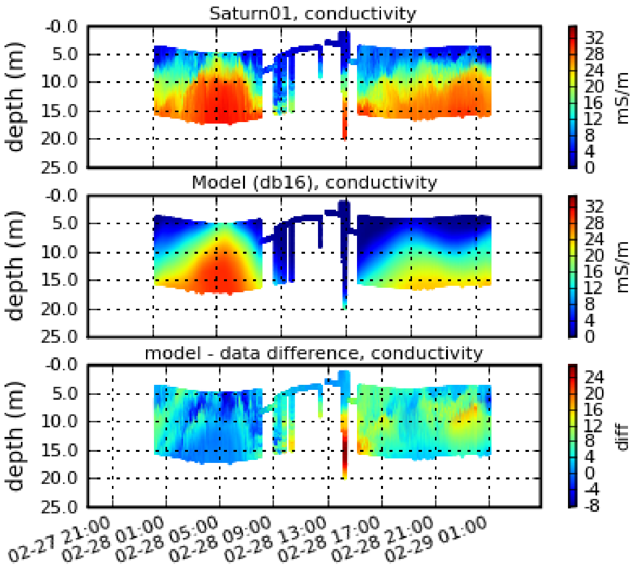


Fig. 2. A data product comparing observed conductivity (top), simulated conductivity (middle) and their difference (bottom) against depth from a vertically mobile platform. In our framework, this kind of static data product becomes *mashable*: parameterized, and reusable in various situational applications.

Scientific mashups push the requirements of enterprise mashups even further. In addition to large datasets and domain-specialization, scientific mashups are relevant to a much broader range of potential customers. Besides highly-specialized domain experts (e.g., the seven people in the world who understand your research the best), students from K12 through post-graduate, collaborators from different fields, the press, the general public, industry colleagues, and policy makers are all potential consumers of scientific results delivered by a mashup. In contrast, an enterprise mashup is intended for a much narrower range of user types — perhaps just one or two specialized analysts or a set of identically-trained customer-service agents. Further, significant scientific findings are intrinsically non-obvious, complicating their exposition. Although a map of addresses can be interpreted by nearly anyone, the meaning of Figure 1 is difficult to ascertain without explanation. (The reader is encouraged to try to interpret the ensemble before reading the explanation in Section 2).

Scientific mashups are also expected to present rather large datasets at one time. For example, the timeseries in Figure 2 displays two days of measurements from a profiling mooring managed by the Center for Coastal Margin Observation and Prediction (CMOP). The sensor climbs up and down within the water column sampling at around 6Hz, generating over 1 million measurements over a two day period. Apprehending the gross features of a million data points requires the higher bandwidth of the human eye — visualization must replace top-k filtering and coarse statistical aggregation at this scale and complexity.

Our work aims to simplify authorship and customization of scientific, visualization-oriented "mashup" applications for diverse audiences.

2 Modeling Scientific Mashups

Our design of a mashup framework suitable for scientific data communication was inspired by the ubiquity of visual *data product ensembles* in scientific discourse.

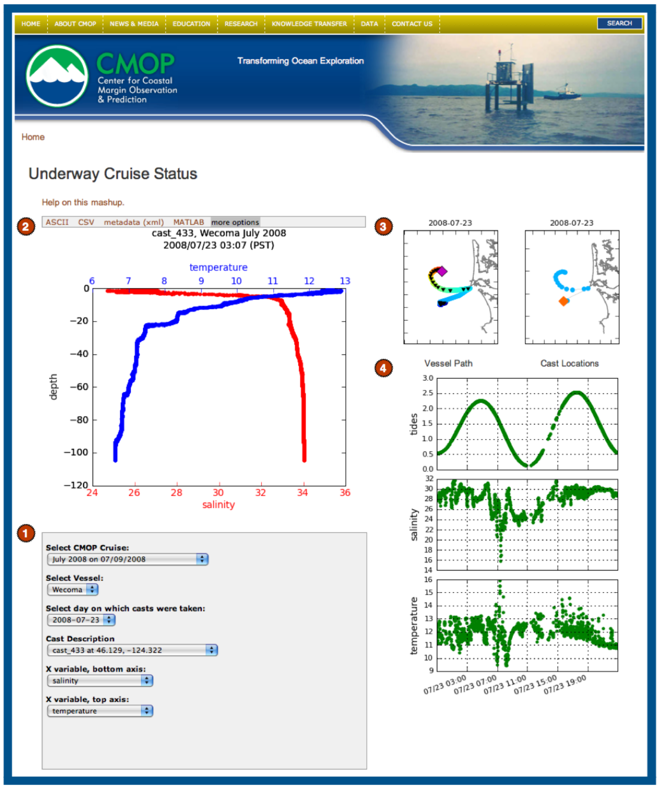
Example: Figure 1 provides an example of an ensemble from a Powerpoint presentation. Each map represents the Columbia River Estuary colored by the water's salinity along the river bottom as computed by the SELFE ocean-circulation model [21]. The large map at the left is the average bottom salinity from 1999-2006. The dense, salty water at the bottom is less dominated by river discharge and tidal influences, and is therefore a better characterization of estuary behavior than other variables such as temperature. The six smaller maps at the right are organized into two columns. Each row is a different year, and only data from the two-month period of December and January is shown for each year. The left column is the average bottom salinity for each of three years: 1999, 2001, 2006. The right column is the average bottom salinity subtracted from the overall average salinity, indicating anomalies. The bar chart at the lower left indicates anomolies for *salinity intrusion length* year-by-year.

This ensemble was constructed from static images by Antonio Baptista, director of CMOP, as part of a presentation on climatological (i.e., long-term) variability. Each static image was generated by the programming staff at CMOP specifically for the presentation. The ensemble was crafted to convey a specific message — that the long-term collection of *hindcast* results generated from the SELFE ocean-circulation model was accurately capturing climatological variability, including known anomalies. Specifically, this ensemble demonstrates that the model captures the fact that 1999 was fresher than average, while 2006 was unremarkable. The year 1999 is known to have exhibited extremely high river discharge in late December, so a fresh estuary agrees with observation.

2.1 Injecting Interactivity

The example ensemble in Figure 1 immediately suggests a family of related ensembles designed to answer related questions: Is the temperature signature anomalous in 1999 as well? Does the 1999 anomaly persist through the Spring freshet in May? This situation illustrates one goal of this research: to allow scientists to re-parameterize and re-purpose this ensemble without relying on programming staff.

To maximize the return on programmer effort, a scientific mashup framework should allow a non-programmer to reuse and repurpose programmer-crafted data products. In contrast, consider workflow systems, which also aim to raise the level of abstraction for authoring data processing pipelines [8, 16] and visual scientific applications [18, 1]. Workflow systems generally support reuse — a workflow authored by one programmer can be accessed by other users of the same platform



Underway Cruise Status

Help on this mashup.

cast_433, Wecoma July 2008
2008/07/23 03:07 (PST)

Select CMOP Cruise: July 2008 on 07/09/2008

Select Vessel: Wecoma

Select day on which casts were taken: 2008−07−23

Cast Description: cast_433 at 46.129, −124.322

X variable, bottom axis: salinity

X variable, top axis: temperature

Fig. 3. A mashup in use at the NSF Science and Technology Center for Coastal Margin Observation and Prediction built to review profile measurements taken during research cruises. The are four main sections: 1) a chain of parameters specifying which visualizations to display, 2) the cast profile plot with two simultaneous variables on the x-axis and depth on the y-axis, 3) a pair of maps providing spatial context for the cast, and 4) A set of timeseries displaying surface measurements gathered from the vessel during the same day.

[8, 16, 18]. However, our stakeholders find the expressive data-flow abstractions adopted by most workflow systems to be only marginally more accessible than general purpose programming languages — and therefore effectively *inaccessible* to non-programmers.

The ensemble in Figure 1 is not designed to be interactive — it conveys a specific scientific message without requiring additional user input. In contrast, consider Figure 3. This ensemble is divided into four areas:

1. **User controls:** Users can select a completed research cruise using the topmost select widget in area (1). The vessel of interest can be chosen using the second select widget. The available vessels depend on the currently chosen cruise. The third and fourth select widgets allow the user to choose the day the cast was taken and particular cast, respectively. The choices available in these selects are dependent on the chosen cruise and vessel. The final two

select widgets allow the user to choose the two x-axis variables that will be displayed in the cast profile image.

2. **Cast profile:** The y-axis of the cast profile is depth, and the configurable top and bottom axes reflect a measured variable.
3. **Daily Map:** The daily maps show vessel activity for the chosen day. The vessel-path map shows the vessel path colored by time of day (red is later, blue is earlier). The pink diamond shows the final vessel location at midnight. The black dots show cast locations. The cast-location map shows the cast locations as blue circles, with the selected cast highlighted in orange. These maps provide *spatial context* needed to interpret the the cast profile. For example, a cast profile near the estuary (as is the cast if Figure 3), then a strong freshwater signal at the surface is expected.
4. **Daily Timeseries:** The timeseries plots show tidal elevations, near-surface salinity, and near-surface temperature from the flow-through sensor package on board the vessel. The tidal elevations provide *temporal context* — the plume is fresher during low tide.

In contrast to the “display-only” mashup of Figure 1, Figure 3 involves user controls for re-parameterizing the mashup. Our mashup model erases the distinction between input and output components, allowing any *mashable* item to both display a set of values to the user, and (optionally) allow the user to select a subset of values to return to the system. Using this simple data model, we are able to express a wide variety of lightweight science applications without requiring mashup developers to learn a broad repertoire of tools. That is, every component is an instance of one underlying class: an *adapted mashable.*

2.2 Inferring Data Flow

Consider the following definitions. We adopt a parameterized type syntax borrowed from the polymorphic features of C++ and Java. Each type variable is a *scheme* — set of attribute names.

$$\begin{aligned}
\text{Environment}\langle E\rangle &:: E \rightarrow \text{list of strings}\\
\text{Domain}\langle T\rangle &:: \text{a relation with attributes } T\\
\text{Mashable}\langle E,T\rangle &:: \text{Environment}\langle E\rangle \rightarrow \text{Domain}\langle T\rangle\\
\text{Adaptor}\langle E,T\rangle &:: \text{Environment}\langle E\rangle \rightarrow \text{Domain}\langle T\rangle \rightarrow \text{Environment}\langle E \cup T\rangle\\
\text{AdaptedMashable}\langle E,T\rangle &:: \text{Environment}\langle E\rangle \rightarrow \text{Environment}\langle E \cup T\rangle
\end{aligned}$$

Each Mashable is a simple function abstraction for web services, database queries, shell programs, etc: given a set of parameters, return a set of tuples. Each formal parameter is a string (e.g.“cruise”, “date”, etc.). Parameter values are *sequences* of strings to allow multi-valued selections (e.g., a range of dates instead of a single date). Individual values are untyped; their interpretation is left up to the consumer, following conventions of REST.

Informally, each adaptor takes a set of tuples, displays them to the user, allows the user to select some subset, and then converts that subset to an environment. The keys of the resulting environment are the attributes of the source relation (as expressed in the definition of Adaptor above). For example, the first select box in area (1) of Figure 3 is populated from a relation with attributes $(cruise, startdate)$. These tuples are converted into a drop down menu by an appropriate adaptor. When the user selects a particular cruise, the adaptor (back on the server, after appropriate translations) receives the user's selection [("July 2007", "7/4/2007")]. The environment passed to the next mashable is the current environment E updated with values from the new environment T with keys $(cruise, startdate)$.

In this model, a *mashup* is a graph where each vertex is an AdaptedMashable (AM). Edges are derived implicitly from parameter dependency information. For example, in Figure 3, one of the select widgets allows the user to choose a particular day of the research cruise. The two maps at the upper right both make use of the selected day value to determine which data to display. No explicit link between the *day* select widget and the two context maps is required. Simply by appearing in the same scope, consumers of the *day* parameter are implicitly dependent on the producer of the *day* parameter. By synchronizing the parameters of each mashable to common sources, we reduce the chance of presenting a dangerously misleading mashup. For example, Figure 1 only make sense if all products pertain to bottom salinity — the system can help enforce this constraint by keeping all products synchronized (unless explicitly overridden by the user — see below).

Edge Inference. More precisely, we heuristically infer an edge between two vertices $X\langle E_X, T_X\rangle$ and $Y\langle E_Y, T_Y\rangle$ if $E_Y \subset T_X$. That is, if one AM supplies all the parameters that another AM needs, then connect them. Next, we infer edges wherever two upstream AMs together can supply the necessary parameters. That is, given a vertex $X\langle E_X, T_X\rangle$, and an edge $(Y\langle E_Y, T_Y\rangle, Z\langle E_Z, T_Z\rangle)$, infer an edge (Z, X) if $E_X \subset T_Y \cup T_Z$. We continue this process for longer paths through the graph until we converge. For example, the plot in area (2) of Figure 3 (call it $P1$) requires values for *cruise*, *vessel*, and *castId*. The first select box $S1$ provides values for *cruise* and *startdate*, the second select box $S2$ requires values for *vessel* (given a *cruise*), and the third $S3$ provides values for *castid* (given a *cruise* and a *vessel*), so we infer a path in the graph $S1 \rightarrow S2 \rightarrow S3 \rightarrow P1$.

Since we rely on an implicit dependency graph among AMs, we must tolerate both *underspecified* and *overspecified* mashups. An underspecified mashup involves AMs that require values for parameters not supplied by any other AMs. We address underspecified mashups by requiring that all mashables either adopt default values for all required parameters or tolerate their omission. For example, the right-hand map plot in area (3) of Figure 3 displays all casts for a given day, but also highlights a particular cast. The author of the underlying adaptor is expected to handle the case where no cast is specified: perhaps no cast is highlighted, or the first cast in the list is highlighted, for example. By requiring that all mashables and tolerate missing parameters, we reduce the number of invalid mashups that can be expressed by the user. Anecdotally, we observe that exceptions and error messages are to be avoided at all costs: Users simply assume that the system is not working and stop using it rather than read the message and correct the problem.

An overspecified mashup provides multiple sources for the same parameter values. For example, the choice of a cast implies a temporal context: the time the cast was taken. However, the choice of a cruise also implies a temporal context: the start time of the cruise itself. A mashable that looks up the tidal elevation based on time must choose between these two timestamps, but there is not necessarily an unambiguous way to do so. In this case, we break the tie by observing that the cast is influenced by the cruise, so the cast time is in a sense more specific — it already takes into account the information supplied by the cruise. We refer to this heuristic as the *path of greatest influence* (PGI). The PGI is simply the backwards path through the directed graph that passes through largest number of competing sources for a parameter. The source to use is the nearest node along the PGI. The algorithm to implement this decision is straightforward: reverse the edges, find the longest path from the target vertex to a vertex supplying the overspecified parameter, and select the first vertex. Ties are currently broken by document order in the HTML — nearest nodes in document order are linked, under the assumption that products are usually added to the mashup in dependency order. Another tie-breaking strategy we are exploring is to automatically multiplex the overspecified product. For example, if two casts are in scope and the designer appends a tidal chart product, there is no unambiguous way to choose between them. Rather than assume the second cast is preferable to first, we can simply insert an additional copy of the tidal chart — one for each cast. This feature is not yet tested.

The implicit dependency graph is one mechanism for specifying data flow, but there are two others. Each mashup is equipped with a *root environment* configured by the mashup designer. The root environment is by default empty, but can be populated with metadata (parameter-value pairs) to resolve ambiguities in the dependency graph or to supply information that cannot be derived anywhere else. For example, the colors in Figure 1 all pertain to bottom salinity in the period 1999-2006. The pairs ($variable = bottomsalinity$, $startyear = 1999$, and $endyear = 2006$) can be inserted into the root environment. The root environment is just another node in the dependency graph — products will still pull their parameters from the nearest upstream source unless explicitly overridden by the designer.

Mashup developers may also individually set parameter values for individual products. Parameters set explicitly for individual products override other sources of parameters and allow fine-tuning of the mashup. For example, the individual years selected for the rows at the right-hand side of Figure 1 are explicitly set by the designer rather than being passed by the dataflow graph.

2.3 Tailoring Mashups for Specific Audiences

Consider the context provided by the static ensemble in Figure 1 for interpreting the underlying data, both explicit and implicit. The title at the upper left gives explicit context: the proper interpretation of the color (bottom salinity) and the overall time period considered (1999-2006). The fact that all the map images pertain to the same geographical region (the Columbia River Estuary), is implicit. The units of the color bar (practical salinity units, or just psu) are also implicit, since no other units are in common usage. The meaning of the y-axis label (SIL stands for Salinity Intrusion Length) is also implicit. Even knowledge of the expanded acronym does not necessarily help a novice interpret the graph. The intrusion length is in meters, but what distance is it measuring? Domain experts will understand that an estuary length is measured along its primary channel, which is usually well-defined for ship traffic, but a novice will find it difficult to develop intuition for the physics without further explanation.

We therefore seek transformations that preserve the scientific meaning of the mashup but tailor it to different audiences. We define three techniques for tailoring mashups: inserting *context products*, changing *application style*, and *re-skinning*.

The first method is to insert new products into the mashup that expose what was going on "nearby" — not just in time and space, but nearby in the overall parameter space. For example, information on tides, weather, daylight, other observational platforms, other models, and so on all potentially enhance interpretation. Additional products that simply display values in the root environment are also applicable here: the title bar in Figure 1 is an example of a simple "product."

The second method is to tailor the application style for different audiences. Anecdotally, we find that experts prefer to fill screen real estate with additional data, but novices prefer to study one product at a time to avoid feeling overwhelmed. The mashup framework supports converting from a dashboard-style interface (all products at once) to a wizard-style interface (one product at at time) without additional programming.

Finally, mashups can be re-purposed for display to different audiences by re-skinning the *mashable* components of the mashup with different *adaptors*. The mashup in Figure 3 depicts temperature as a variable on the x-axis of a cast-profile plot. This succinct approach to data visualization is desirable when the intended audience is a group of domain experts. However, this visualization is likely difficult to understand and un-engaging for elementary-school science students. In order to address this issue, the original mashup can be *re-skinned* to appeal to a wider audience. As shown in Figure 4, the product which displays depth on the y-axis and temperature and salinity on the x-axis is replaced by a product which displays depth as an animation of a cast profiling device beneath the vessel moving up and down in the water column, with the corresponding temperature and salinity displayed as familiar graphical thermometers. In this fashion, re-skinning makes the same data product accessible to users of widely differing backgrounds and skill sets.

The discussion in this section exposes three steps in creating a scientific mashup:

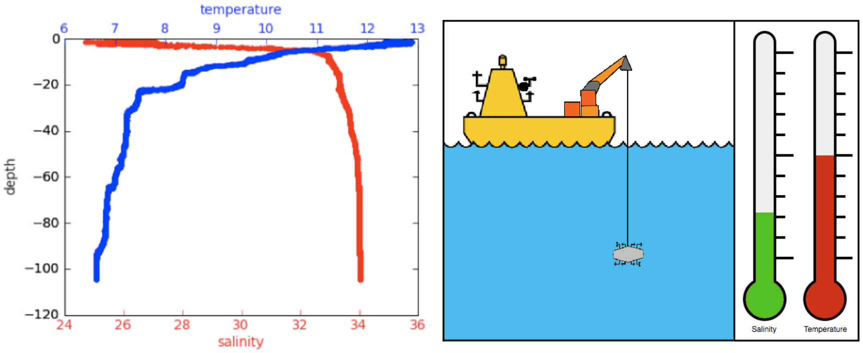


Fig. 4. Different adaptors can present data from the same mashable in different ways. The adaptor on the left displays depth along the y-axis and temperature and salinity along the x-axis of the cast plot. The adaptor on the right illustrates the dependency between temperature and salinity and depth by allowing the user to drag the cast probe up and down within the water column in order to see the change in temperature and salinity associated with the change in depth.

1. **Wrap** Programmers must wrap each data source as a mashable accepting an arbitrary environment of parameters mapped to sequences of values and returning a set of tuples.
2. **Synch** Mashup authors (usually non-programmers) choose from a set of *adapted mashables* — each one a mashable with a specific visual “skin.” The set of selected mashables are wired together as a data flow graph derived using heuristics involving document order, implicit dependency information, and user input.
3. **Tailor** Mashups can be re-purposed for different audiences by attaching additional adapted mashables, using different adaptors for the same mashables, or re-organizing into a different application style (i.e., a dense, AJAX-powered single-screen application vs. a wizard-style, question-and-answer-oriented application).

2.4 Challenges and Limitations

The challenge of the Scientific Mashup problem is to support a broadly functional class of web applications without incurring the cognitive load associated with traditional programming. We are experimenting with an extremely simple conceptual model: mashables for retrieving data, adaptors for interacting with the user, and sequences of strings for data flow. As a result of this design choice, we rely heavily on the functionality of the mashables.

By design, we do not permit additional operations between adaptors and mashables, such as filtering, data cleaning, type conversion, or arithmetic. We assume that all such work is done inside the mashable. We impose no restrictions on the expressiveness of the mashable internals. We anticipate that some mashable components will execute complex scientific workflows to generate the domain, or otherwise execute arbitrary programs. However, because we are agnostic to the language or system used to author mashables, our work is complementary to research in programming languages, scientific workflows, and data integration systems. We are interested in empowering the non-programmer to craft interactive scientific ensembles using basic building blocks. We are exploring the question, "What are the limits to the applications can we provide assuming the user is only willing to specify 1) the visualizations they are interested in, 2) their arrangement on-screen, 3) a few parameter values in a root environment?"

Our reliance on a simplified relational model represents another limitation: data sources that return XML must be flattened into relations before they can be used with our system. We have found XML to be rather unpopular among scientific programmers for simple tasks, and we want to keep the barrier to entry for our mashable authors as low as possible.

3 Related Work

Workflow systems attempt to raise the level of abstraction for scientific programmers by adding language features: visual programming, provenance, limited task parallelism, fault tolerance, type-checking, sophisticated execution models [3, 8, 16, 18]. In contrast, we adopt a top-down approach: begin with a static data product ensemble, then endow it with interactivity and publish it online.

The VisTrails system has a suite of advanced features useful as mashup support services. VisTrails exploits the graph structure of the workflow and a database of existing workflows to provide a higher-level interface for workflow composition. Users can create new workflows "by analogy" to existing workflows and the system can help "autocomplete" a workflow by offering suggestions based on the graph structure [14]. Both features rely on graph matching with an existing corpus of workflows . VisTrails also adopts a spreadsheet metaphor for displaying and browsing related visualizations. A series of workflow executions can be compared side-by-side in the cells of a grid. Further, a series of relaetd images can be generated in one step using a parameter exploration — an iterative execution of the same workflow across a range of parameter values. finally, the VisTrails system also allows an individual workflow to be compiled into a simple web application[19]. These features all help reduce the programmer effort required to create and re-purpose workflows. Our approach is complementary — we explore the configuration space of multiple synchronized visualizations, while remaining agnostic to how the visualizations are created.

Marini et al. describe a system to publish workflows to the web as interactive applications [9] and corroborate our emphasis on domain-specific solutions [5], but do not consider multiple synchronized workflows, nor runtime-configuration for different audiences.

There exist many commerce- and web-oriented mashup development frameworks freely available for public use [6, 12, 20]. From among the systems available to us we chose to examine Yahoo Pipes and Microsoft's Popfly in detail.

Yahoo Pipes is a web-based mashup solution. Pipes allows users to compose existing data feeds into new feeds that can be made available on the web. The Pipes system appears to be well suited to this task, but requires that a user be at least passingly familiar with standard Internet feed formats and XML concepts. Pipes does attempt to allow users to integrate less-structured information from arbitrary web pages into its mashups; this integration appears to be limited to displaying the contents of a page (or portion thereof) alongside structured feed information. In contrast, our requirements demand that users be able to draw from a wide range of scientific data without being forced to understand the particular storage format of that data.

Popfly is Microsoft's offering for a mashup-development framework. Popfly differentiates itself from other mashup solutions by offering a very rich user interface. Of particular interest is the feature that allows a user to examine a mashup component at different levels of abstraction. At the simplest level, a mashup component is presented as a graphical control with inputs and outputs that can be connected via drag and drop operations, similarly to Yahoo Pipes. At the next level of detail, a mashup component is presented as a choice of operations that can be performed on the inputs — integer addition, string concatenation, filtering, etc. At the lowest level of detail, a user is invited to directly edit the javascript code behind a mashup component. In this fashion, different designers with varying levels of programming expertise can interact with Popfly in the fashion that best suits their needs and abilities — all within the same development interface. This ability to drill down to a preferred level of programming-interface complexity is well-suited to a scientific mashup application where a substantial number of users can be expected to have advanced mathematical skills as well as a reasonably advanced programing skill set. However, data processing in Popfly is performed in client-side javascript, which is inefficient in the context of the large data sets and computationally expensive operations inherent in a scientific mashup.

We observe that software vendors are increasingly aware of audience-adaptibility. Microsoft Office products hide unused options from their menus in order to avoid overwhelming novice users. Tax preparation software (e.g. TurboTax [17]) provide multiple application styles for the same core content. A questionnaire application style (called EasyStep in TurboTax) can be used in conjunction with a form-oriented application style designed for experts. We are studying how these techniques may be adopted for scientific mashups.

4 System architecture

Our initial mashup system is comprised of five major components. The environment, the mashable, the domain, the adaptor, and the mashup engine.

1. **Environment** An environment is a map of keys to values and is the input to a mashable. All mashups are designed to accommodate an empty or default environment. Thus a valid product ensemble is guaranteed even in the absence of user interaction.
2. **Mashable** The mashable is the primary means of data-source abstraction. A mashable takes an environment as input and produces a domain as output. Mashables are reusable components created by programmers for end users to work with during runtime configuration of a mashup.
3. **Domain** A domain is a set of tuples returned by a mashable.

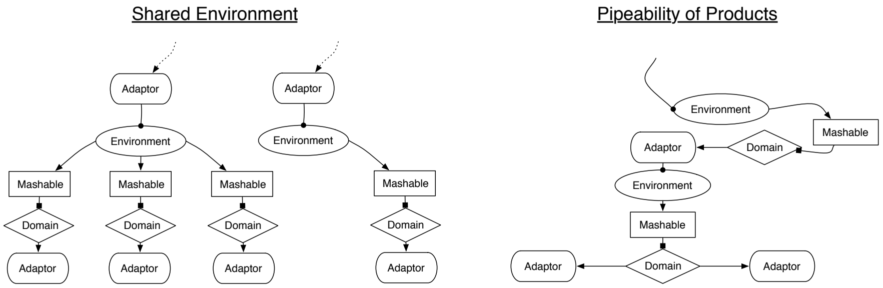


Fig. 5. Possible mashup graph configurations.

4. **Adaptor** Adaptors render domains. All user-interface components are adaptors. Adaptors can be display-only or interactive. Interactive adaptors allows users to modify the environment passed to subsequent mashables in the mashup graph. Like mashables, adaptors are created by programmers for configuration within a mashup by end users. Different adaptors may be associated with a single mashable allowing audience-specific presentations of data.
5. **Mashup Engine** The mashup engine is the context within which the other mashup components exist. Designers browse for mashables and adaptors which have been previously defined and link them together at runtime within the mashup engine. The mashup engine is responsible for automatically synchronizing the mashup components when a user alters the state of an environment via interaction with an adaptor.

It is possible to have multiple products share a single environment, as illustrated in Figure 5 (left). Constructing a mashup in this fashion allows user interaction with a single Adaptor to effect the state of all products that are members of the mashup tree rooted at the shared environment. Products located within a separate subtree of the mashup are not affected by this adaptor interaction. A users interaction with a given product within a product ensemble effects the state of the environment. A cloned instance of the environment is passed down from product to product as in Figure 5 (right). In this way the result of interaction with one product in the chain is passed down to subsequent products in the product chain.

Listing 1.1 provides an example of the mechanisms used to wire a set of mashables and adaptors together in order to create a functioning mashup. In this example the mashup wiring is shown explicitly. However, similar wiring might just as easily take place at runtime as the result of a users interaction with the system.

In line 1, a new instance of a mashup is created. The name (CmopTest) of the mashup serves to identify this particular mashup within a system that may host many different mashup instances at the same time. In lines 3 and 4, A Mashable reference to the CmopVesselMashable is obtained and a new SelectAdaptor is created and named. The CmopVesselMashable is an implementation of Mashable which abstracts vessel specific data within the cmop database. The SelectAdaptor is an adaptor implementation which renders its adapted domain

Listing 1.1. A mashup example linking two select widgets

```
1  Mashup m = new MashupImpl(“CmopTest”);

3  Mashable vessel = CmopVesselMashable.getInstance();
4  Adaptor selectvessel = new SelectAdaptor(“vessel”);
5  Map<String, String>  vesselmap = new HashMap<String, String>();
6  vesselmap.put(SelectAdaptor.KEY, “vessel”);
7  m.addMashable(vessel);
8  m.linkAdaptor(selectvessel, vessel, vesselmap);

10 Mashable cruise = CmopCruiseMashable.getInstance();
11 Adaptor selectcruise = new SelectAdaptor(“cruise”);
12 Map<String, String> cruisemap = new HashMap<String, String>();
13 cruisemap.put(SelectAdaptor.KEY, “cruise”);
14 m.addMashable(cruise);
15 m.linkAdaptor(selectcruise, cruise, cruisemap);

17 m.linkMashable(cruise, vessel);
```

as an html select widget. In lines 5 and 6, we define a mapping to associate the attributes required by the adaptor with attributes available within the domain produced by the adapted mashable. In lines 7 and 8, the mashable instance is added to the mashup and linked to the configured adaptor.

In lines 10-15, a new mashable and adaptor are linked together and added to the mashup. In this case, the mashable is an abstraction of cmop cruise data and the adaptor is again an instance which renders as an html select widget.

Finally, in line 17, the two mashables are linked or ”mashed” together within the context of the mashup. In this case the cruiseMashable is the ”masher” and the vesselMashable is the “mashee”. As such, changes to the cruiseSelectAdaptor made by the user will modify the environment provided to the vesselMashable. This changed environment will cause a corresponding change to the domain produced by the vesselMashable and ultimately to the data displayed to the user by the vesselSelectAdaptor.

5 A Mashup Factory for an Ocean Observatory

In Section 4, we described an initial implementation of the complete mashup model. In this section, we describe an earlier incarnation of these ideas called the *Product Factory.* The Product Factory is currently deployed at the Center for Coastal Margin Observation and Prediction, and is part of the infrastructure for the upcoming Pacific FishTrax website sponsored by the project for Collaborative Research on Oregon Ocean Salmon (CROOS) [13]. The design goals of the Product Factory are:

1. Replace static images with dynamic, user-configurable data products.
   Method: *Distill each product to a set of parameters, an SQL statement, and a short plotting script. All other boilerplate code is provided by the factory.*

2. Simplify the creation of web applications involving multiple data products. **Method:** *Expose each factory product as a RESTful web service, allowing data products, parameter values, and source data to be embedded in HTML without additional server-side programming.*
3. Provide public access to the underlying data, not just the visualization. **Method:** *Registering a product in the factory automatically establishes a web service for data access in a variety of common formats.*

To create a new product, a programmer writes a *product specification* in either XML or Python. An example XML product specification appears in Listing 1.2. The specification involves three sections: a set of parameters (lines 3-15), an *`extractwith`* clause indicating how to extract data (lines 16-21), and a *`plotwith`* clause indicating how to render the data (lines 22-28).

The Product Factory extracts data and parameter domains from a relational database. Generalization of the Factory to allow access to arbitrary data sources was an important motivation in designing the architecture of Section 4.

Parameters have a type that determines their behavior. The base type `Parameter` allows unconstrained user input, and is rendered as a simple text box. A `SelectParameter` (lines 13-15) takes a comma-delimited list of strings and forces the use to select one using an HTML select tag. A `SQLSelectParameter` (e.g., lines 3-5) is similar to a `SelectParameter`, but the choices are drawn from a database using an SQL statement. SQL-powered parameters may depend on the values of earlier parameters. Syntactically, the SQL statement may include embedded placeholders using Python string formatting conventions. In the HTML interface, we use asynchronous javascript to dynamically refresh the values of downstream parameters when upstream parameters change.

Other parameter types available include `MultiSelect` and `SQLMultiSelect` versions that allow multiple option to be selected by the user, `HiddenParameter` that computes a value for downstream processing but does not interact with the user, and `DynamicDefaultParameter` that computes an initial value from upstream parameters but allows arbitrary user input. Each SQL-driven parameters is similar to the product itself — data is extracted from the database and displayed to the user. The observation that parameters are not fundamentally different from products led to their unification in the current model (Section 2).

Like the SQL-powered parameters, the `extractwith` clause uses string-substitution placeholders to reference parameter values. The `plotwith` clause is simply python code executed in an environment with all parameters and plotting libraries pre-loaded as local variables. Most `plotwith` are remarkably short, using just a few MATLAB-style calls provided by the matplotlib 2D plotting library [10]. Since many of our programmers are familiar with MATLAB, matplotlib provides an attractive alternative for programming visualizations.

Each parameter type specifies how to compute a default value. For the base `Parameter` type, the default value is given explicitly in the definition. Parameters with explicit domains use the first value in the domain as the default value. If their domain is empty (e.g., the SQL query returns no records), then a sentinel value akin to NULL is returned. A downstream parameter may or may not be designed to tolerate NULL — if an exception is raised, the downstream

Listing 1.2. A factory product specification in XML. The Product Factory is an early implementation of a scientific mashup framework focused on reducing the code required to publish an interactive data product.

```
<specification>
 <product name=‘‘castprofile”>
  <parameter name=‘‘vessel” type=‘‘SQLSelectParameter”>
    SELECT vessel FROM cruise.vessel
  </parameter>
  <parameter name=‘‘cruise” type=‘‘SQLSelectParameter”>
    SELECT cruise FROM cruise.cruise WHERE vessel=’%(vessel)s’
  </parameter>
  <parameter name=‘‘cast” type=‘‘SQLSelectParameter”>
    SELECT distinct castid, castdescription FROM ctdcast
     WHERE vessel = ’%(vessel)s’ AND cruise = ’%(cruise)s’
  </parameter>
  <parameter name=”variable” type=”SelectParameter”>
   salinity, conductivity, temperature, pressure, turbidity
  </parameter>
  <extractwith>
    SELECT time, −depth as depth, %(variable)s as variabledata
      FROM castobservation
     WHERE vessel = ’%(vessel)s’
       AND cruise = ’%(cruise)s’ AND castid = ’%(cast)s’
  </extractwith>
  <plotwith>
  title(”%s %s %s’ % (cast,vessel,cruise), size=’small’)
  scatter(variabledata, depth, faceted=False)
  xlabel(variable, size=’small’)
  ylabel(’depth (m)’, size=’small’)
  </plotwith>
 </product>
</specification>
```

parameter is itself assigned NULL. This aggressive propagation of NULL values is necessary to prevent exception messages and errant behavior for the end user — in our experience, a product that returns an empty dataset is tolerable, but error messages are not. However, masking exceptions complicates debugging, so we provide a command line tool for testing products in a controlled environment.

The Product Factory simplifies the Wrap step of mashup authorship by changing the skillsets required to publish interactive data products on the web. Instead of learning a host of languages and configuration tools, programmers can simply write a series of related SQL statements and a short MATLAB style-script — higher-level language skills that are generally easier to learn (and that CMOP programmers happen to already possess!)

To register a specification with the Factory, the XML script is uploaded through either a command-line tool or a web form. The script is first tested in the empty environment (all products must produce meaningful results with no user decisions), then loaded into the factory database. Once loaded, rendered products, source data, parameter domains, and other information are all available through a RESTful web service interface. Factory calls are of the form

```
http://server.com?request=<r>&product=<prd>&<p1>=<v1>&...
```

where `r` is one of `getproduct`, `getdata`, `getdomain`, `getspec`, `prd` is the name of a product, `pi` is a parameter name and `vi` is a url-encoded value.

To construct a synchronized ensemble from individual products as in Figure 3, a mashup designer need only include each relevant product in the HTML page using the syntax `<div class="factory" product=‘‘cast"/>` and include the factory javascript library. Each product will be expanded into either an HTML form or a simple image tag, depending on whether or not their parameters need to be displayed. For example, in Figure 3, areas (1) and (2) are together an HTML form rendered for the cast profile product (similar but not identical to Listing 1.2). Any other products that share these parameters can "borrow" from the castprofile product, thereby avoiding the need to have the user specify the relevant cruise, vessel, and cast multiple times. This mechanism supports the Synch step of mashup authorship — product "building blocks" can be brought into the same scope, and the system will derive a wiring diagram for them based on the dependency graph between parameters and products.

6 Conclusions and Future Work

The adoption of a mashup-style of application development is potentially transformative for scientific communication. Faced with exploding data volumes, enormous data heterogeneity, and limited programming staff, development of new data product ensembles is becoming the bottleneck to dissemination of results. Our model unifies visualizations with interactive user controls, providing a simple underlying model that can express a wide-variety of scientific applications. The ability to adapt existing mashups to tailor them for new audiences provides another dimension of reuse. The initial deployment of the Product Factory provides evidence that a successful mashup framework should not just raise the level of abstraction for individual data products, but also provide tools for organizing constituent products into interactive data ensembles. With this approach, scientists are empowered to reuse programmers' work (via support for product synchronization), and the end users are empowered to reuse a scientists' work (via interactive user controls).

Future work includes a formalization of the context model outlined in Section 2. We hope to be able to quantify interpretability based on domain-specific models of context. Equipping the mashup framework with semantics may allow a broader and more useful consideration of context and interpretability. Endowing individual mashable components with semantics will also allow more advanced reasoning by the system. The current implementation relies too heavily on advanced programming skillsets. Mashup designers must at least write HTML to connect mashable components together, but we plan a drag-and-drop interface similar to Microsoft's Popfly or Yahoo Pipes.

Acknowledgements

We wish to thank Antonio Baptista, Roger Barga, Juliana Friere, and Emanuele Santos for helpful discussions. This work was supported by funding from the NSF STC for Coastal Margin Observation and Prediction, a Jim Gray Seed Grant from Microsoft, and the eScience Institute at the University of Washington.

References

[1] Barga, R., Jackson, J., Araujo, N., Guo, D., Gautam, N., Grochow, K., Lazowska, E.: Trident: Scientific Workflow Workbench for Oceanography. In: IEEE Congress on Services, 465–466, IEEE Computer Society, Los Alamitos, CA, USA (2008).

[2] CrimeMapping (2008), `http://www.crimemapping.com/`.

[3] Deelman, E., Singh, G., Su, M.H., Blythe, J., Gil, Y., Kesselman, C., Mehta, G., Vahi, K., Berriman, G.B., Good, J., Laity, A., Jacob, J.C., Katz, D.S.: Pegasus: a Framework for Mapping Complex Scientific Workflows onto Distributed Systems. Scientific Programming Journal 13, 3, 219–237 (2005).

[4] Gray, J., Szalay, A.S.: ”Where the Rubber Meets the Sky: Bridging the Gap between Databases and Science”. IEEE Data Eng. Bull. 27, 4, 3–11 (2004).

[5] Hill, D.J., Minsker, B., Liu, Y., Myers, J.: End-to-End Cyberinfrastructure for Real-Time Environmental Decision Support. In: IEEE eScience (2008).

[6] JackBe. `http://www.jackbe.com`.

[7] Jhingran, A.: Enterprise Information Mashups: Integrating Information, Simply. In: 32nd International Conference on Very Large Data Bases (2006).

[8] The Kepler Project. `http://kepler-project.org`.

[9] Marini, L., Kooper, R., Bajcsy, P., Myers, J.D.: Publishing Active Workflows to Problem-Focused Web Spaces. In: IEEE eScience (2008).

[10] The Matplotlib Library. `http://matplotlib.sourceforge.net`.

[11] Mint.com. `http://mint.com`.

[12] Microsoft Popfly. `http://www.popfly.com`.

[13] Collaborative Research on Oregon Ocean Salmon project (ProjectCROOS). `http://projectcroos.com`.

[14] Silva, C., VisTrail, personal communication (2008).

[15] SnapTweet. `http://snaptweet.com/`.

[16] The Taverna Project. `http://taverna.sourceforge.net`.

[17] TurboTax. `http://turbotax.intuit.com`.

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

[19] Santos, E., Freire, J., Silva, C.: Using Workflow Medleys to Streamline Exploratory Tasks. `http://www.research.ibm.com/gvss/2007/presentations/emanuele_ibm_gvss2007.pdf`.

[20] Yahoo Pipes. `http://pipes.yahoo.com/pipes`.

[21] Zhang, Y.L., Baptista, A.M.: SELFE: A semi-implicit Eulerian-Lagrangian finite-element model for cross-scale ocean circulation. Ocean Modelling 21, 3-4, 71–96 (2008).