The Architecture of Open Source Applications

Amy Brown & Greg Wilson

Contents

Chapter 1

VisTrails

VisTrails[1] is an open-source system that supports data exploration and visualization. It includes and substantially extends useful features of scientific workflow and visualization systems. Like scientific workflow systems such as Kepler and Taverna, VisTrails allows the specification of computational processes which integrate existing applications, loosely-coupled resources, and libraries according to a set of rules. Like visualization systems such as AVS and ParaView, VisTrails makes advanced scientific and information visualization techniques available to users, allowing them to explore and compare different visual representations of their data. As a result, users can create complex workflows that encompass important steps of scientific discovery, from data gathering and manipulation to complex analyses and visualizations, all integrated in one system.

A distinguishing feature of VisTrails is its provenance infrastructure [3]. VisTrails captures and maintains a detailed history of the steps followed and data derived in the course of an exploratory task. Workflows have traditionally been used to automate repetitive tasks, but in applications that are exploratory in nature, such as data analysis and visualization, very little is repeated—change is the norm. As a user generates and evaluates hypotheses about their data, a series of different, but related, workflows are created as they are adjusted iteratively.

VisTrails was designed to manage these rapidly-evolving workflows: it maintains provenance of data products (e.g., visualizations, plots), of the workflows that derive these products, and their executions. The system also provides annotation capabilities so users can enrich the automatically-captured provenance.

Besides enabling reproducible results, VisTrails leverages provenance information through a series of operations and intuitive user interfaces that help users to collaboratively analyze data. Notably, the system supports reflective reasoning by storing temporary results, allowing users to examine the actions that led to a result and to follow chains of reasoning backward and forward. Users can navigate workflow versions in an intuitive way, undo changes without losing results, visually compare multiple workflows and show their results side-by-side in a visualization spreadsheet.

VisTrails addresses important usability issues that have hampered a wider adoption of workflow and visualization systems. To cater to a broader set of users, including many who do not have programming expertise, it provides a series of operations and user interfaces that simplify workflow design and use [3], including the ability to create and refine workflows by analogy, to query workflows by example, and to suggest workflow completions as users interactively construct their workflows using a recommendation system [4]. We have also developed a new framework that allows the creation of custom applications that can be more easily deployed to (non-expert) end users.

[1] `http://www.vistrails.org`

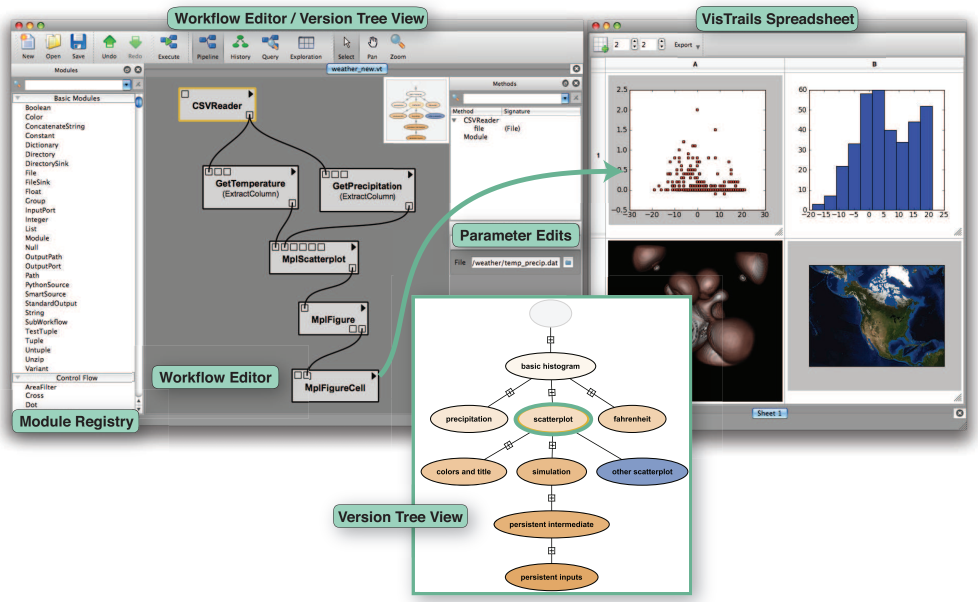


Figure 1.1: Components of the VisTrails user interface

The extensibility of VisTrails comes from an infrastructure that makes it simple for users to integrate tools and libraries, as well as to quickly prototype new functions. This has been instrumental in enabling the use of the system in a wide range of application areas, including environmental sciences, psychiatry, astronomy, cosmology, high-energy physics, quantum physics, and molecular modeling.

To keep the system open-source and free for all, we have built VisTrails using only free, open-source packages. VisTrails is written in Python and uses Qt as its GUI toolkit (through PyQt Python bindings). Because of the broad range of users and applications, we have designed the system from the ground up with portability in mind. VisTrails runs on Windows, Mac and Linux.

1.1 System Overview

Data exploration is an inherently creative process that requires users to locate relevant data, to integrate and visualize this data, to collaborate with peers while exploring different solutions, and to disseminate results. Given the size of data and complexity of analyses that are common in scientific exploration, tools are needed that better support creativity.

There are two basic requirements for these tools that go hand in hand. First, it is important to be able to specify the exploration processes using formal descriptions, which ideally, are executable. Second, to reproduce the results of these processes as well as reason about the different steps followed to solve a problem, these tools must have the ability to systematically capture provenance. VisTrails was designed with these requirements in mind.

1.1.1 Workflows and Workflow-Based Systems

Workflow systems support the creation of pipelines (workflows) that combine multiple tools. As such, they enable the automation of repetitive tasks and result reproducibility. Workflows are rapidly replacing primitive shell scripts in a wide range of tasks, as evidenced by a number of workflow-based applications, both commercial (e.g., Apple's Mac OS X Automator and Yahoo! Pipes) and academic (e.g., NiPype, Kepler, and Taverna).

Workflows have a number of advantages compared to scripts and programs written in high-level languages. They provide a simple programming model whereby a sequence of tasks is composed by connecting the outputs of one task to the inputs of another. Figure 1.1 shows a workflow which reads a CSV file that contains weather observations and creates a scatter plot of the values.

This simpler programming model allows workflow systems to provide intuitive visual programming interfaces, which make them more *suitable for users who do not have substantial programming expertise*. Workflows also have an *explicit structure*: they can be viewed as graphs, where nodes represent processes (or modules) along with their parameters and edges capture the flow of data between the processes. In the example of Figure 1.1, the module `CSVReader` takes as a parameter a file name (`/weather/temp_precip.dat`), reads the file, and feeds its contents into the modules `GetTemperature` and `GetPrecipitation`, which in turn send the temperature and precipitation values to a matplotlib function that generates a scatter plot.

Most workflow systems are designed for a specific application area. For example, Taverna targets bioinformatics workflows, and NiPype allows the creation of neuroimaging workflows. While VisTrails supports much of the functionality provided by other workflow systems, it was designed to support general exploratory tasks in a broad range of areas, integrating multiple tools, libraries, and services.

1.1.2 Data and Workflow Provenance

The importance of keeping provenance information for results (and data products) is well recognized in the scientific community. The provenance (also referred to as the audit trail, lineage, and pedigree) of a data product contains information about the process and data used to derive the data product. Provenance provides important documentation that is key to preserving the data, to determining the data's quality and authorship, and to reproducing as well as validating the results [2].

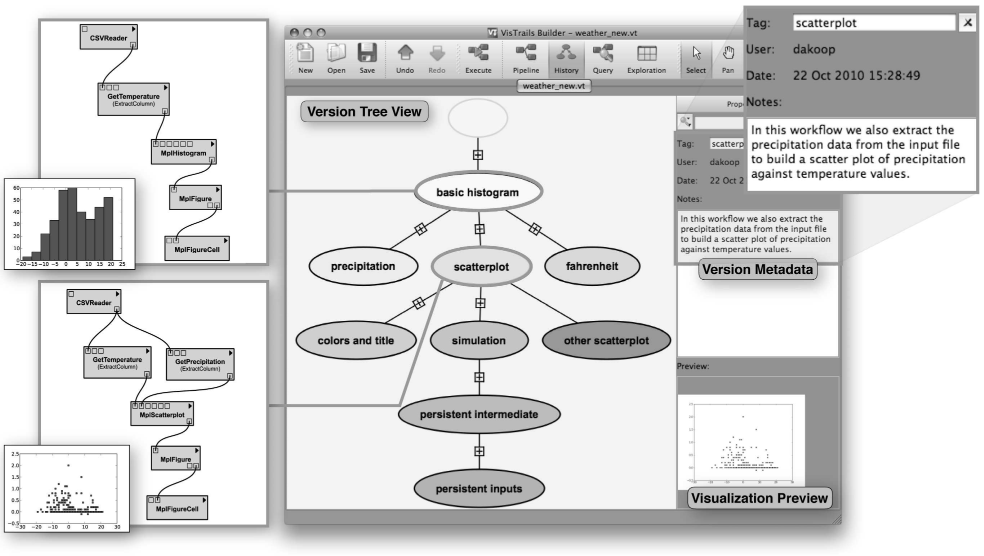
An important component of provenance is information about *causality*, i.e., a description of a process (sequence of steps) which, together with input data and parameters, caused the creation of a data product. Thus, the structure of provenance mirrors the structure of the workflow (or set of workflows) used to derive a given result set.

In fact, a catalyst for the widespread use of workflow systems in science has been that they can be easily used to automatically capture provenance. While early workflow systems have been *extended* to capture provenance, VisTrails was *designed* to support provenance.

1.1.3 User Interface and Basic Functionality

The different user interface components of the system are illustrated in Figure 1.1 and Figure 1.2. Users create and edit workflows using the Workflow Editor. To build the workflow graphs, users can drag modules from the Module Registry and drop them into the Workflow Editor canvas. VisTrails provides a series of built-in modules, and users can also add their own (see Section 1.3.4 for details). When a module is selected, VisTrails displays its parameters (in the Parameter Edits area) where the user can set and modify their values.

As a workflow specification is refined, the system captures the changes and presents them to the user in the Version Tree View described below. Users may interact with the workflows and their



Tag: scatterplot
User: dakoop
Date: 22 Oct 2010 15:28:49
Notes:
In this workflow we also extract the precipitation data from the input file to build a scatter plot of precipitation against temperature values.

Figure 1.2: The provenance of exploration is shown in the version tree and enriched by annotations

results in the VisTrails Spreadsheet. Each cell in the spreadsheet represents a view that corresponds to a workflow instance. In Figure 1.1, the results of the workflow shown in the Workflow Editor are displayed on the top-left cell of the spreadsheet. Users can directly modify the parameters of a workflow as well as synchronize parameters across different cells in the spreadsheet.

The Version Tree View helps users to navigate through the different workflow versions. As shown in Figure 1.2, by clicking on a node in the version tree, users can view a workflow, its associated result (Visualization Preview), and metadata. Some of the metadata is automatically captured, e.g., the id of the user who created a particular workflow and the creation date, but users may also provide additional metadata, including a tag to identify the workflow and a written description.

1.2 Project History

Initial versions of versions of VisTrails were written in Java and C++ [1]. The C++ version was distributed to a few early adopters, whose feedback was instrumental in shaping our requirements for the system.

Having observed a trend in the increase of the number of Python-based libraries and tools in multiple scientific communities, we opted to use Python as the basis for VisTrails. Python is quickly becoming a universal modern glue language for scientific software. Many libraries written in different languages such as Fortran, C, and C++ use Python bindings as a way to provide scripting capabilities. Since VisTrails aims to facilitate the orchestration of many different software libraries in workflows, a pure Python implementation makes this much easier. In particular, Python has dynamic code loading features similar to the ones seen in LISP environments, while having a much bigger developer community, and an extremely rich standard library. Late in 2005, we started the development of the current system using Python/PyQt/Qt. This choice has greatly simplified extensions to the system, in particular, the addition of new modules and packages.

A beta version of the VisTrails system was first released in January 2007. Since then, the system has been downloaded over twenty-five thousand times.

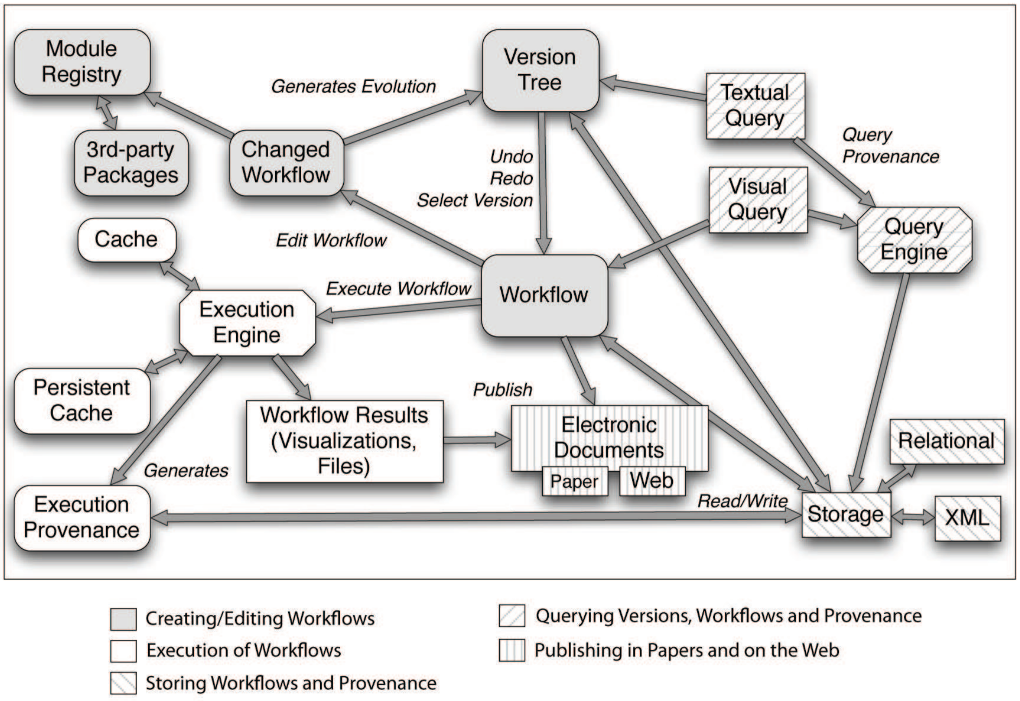


Figure 1.3: VisTrails Architecture

1.3 Inside VisTrails

The internal components that support the user-interface functionality described above are depicted in the high-level architecture of VisTrails, shown in Figure 1.3. Workflow execution is controlled by the Execution Engine, which keeps track of invoked operations and their respective parameters and captures the provenance of workflow execution (Execution Provenance). As part of the execution, VisTrails also allows the caching of intermediate results both in memory and on disk. As we discuss in Section 1.3.2, only new combinations of modules and parameters are re-run, and these are executed by invoking the appropriate functions from the underlying libraries (e.g., matplotlib). Workflow results, connected to their provenance, can then be included in electronic documents (Section 1.4.6).

Information about changes to workflows is captured in a Version Tree, which can be persisted using different storage backends, including an XML file store in a local directory and a relational database. VisTrails also provides a query engine that allows users to explore the provenance information.

We note that, although VisTrails was designed as an interactive tool, it can also be used in server mode. Once workflows are created, they can be executed by a VisTrails server. This feature is useful in a number of scenarios, including the creation of Web-based interfaces that allows users to interact with workflows and the ability to run workflows in high-performance computing environments.

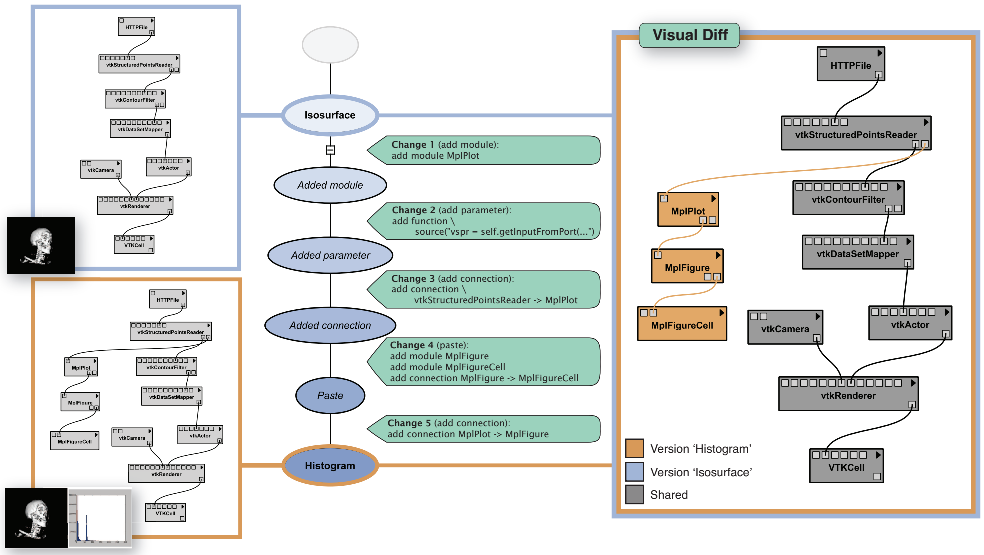

Isosurface
Change 1 (add module): add module MplPlot
Added module
Change 2 (add parameter): add function \ source("vspr = self.getInputFromPort(...")
Added parameter
Change 3 (add connection): add connection \ vtkStructuredPointsReader -> MplPlot
Added connection
Change 4 (paste): add module MplFigure add module MplFigureCell add connection MplFigure -> MplFigureCell
Paste
Change 5 (add connection): add connection MplPlot -> MplFigure
Histogram
Visual Diff
HTTPFile
vtkStructuredPointsReader
MplPlot
vtkContourFilter
MplFigure
vtkDataSetMapper
MplFigureCell
vtkCamera
vtkActor
vtkRenderer
VTKCell
🟧 Version 'Histogram'
🟦 Version 'Isosurface'
⬛ Shared

Figure 1.4: The change-based provenance model encodes versions as sequences of changes (center). This model makes it easy to compare different version of workflows (right)

1.3.1 The Version Tree: Change-Based Provenance

A new concept we introduced with VisTrails is the notion of provenance of workflow evolution [3]. In contrast to previous workflow and workflow-based visualization systems, which maintain provenance only for derived data products, VisTrails treats the workflows as first-class data items and also captures their provenance. The availability of workflow-evolution provenance supports reflective reasoning. Users can explore multiple chains of reasoning without losing any results, and because the system stores intermediate results, users can reason about and make inferences from this information. It also enables a series of operations which simplify exploratory processes. For example, users can easily navigate through the space of workflows created for a given task, visually compare the workflows and their results (see Figure 1.4), and explore (large) parameter spaces. In addition, users can query the provenance information and learn by example.

The workflow evolution is captured using the change-based provenance model. As illustrated in Figure 1.4, VisTrails stores the operations or changes that are applied to workflows (e.g., the addition of a module, the modification of a parameter, etc.), akin to a database transaction log. This information is modeled as a tree, where each node corresponds to a workflow version, and an edge between a parent and a child node represents the change applied to the parent to obtain the child. We use the terms version tree and vistrail (short for *visual trail*) interchangeably to refer to this tree. Note that the change-based model uniformly captures both changes to parameter values and to workflow definitions. This sequence of changes is sufficient to determine the provenance of data products and it also captures information about how a workflow evolves over time. The model is both simple and compact—it uses substantially less space than the alternative of storing multiple *versions* of a workflow.

There are a number of benefits that come from the use of this model. Figure 1.4 shows the visual difference functionality that VisTrails provides for comparing two workflows. Although the workflows are represented as graphs, using the change-based model, comparing two workflows becomes very simple: it suffices to navigate the version tree and identify the series of actions required to transform one workflow into the other.

Another important benefit of the change-based provenance model is that the underlying version tree can serve as a mechanism to support collaboration. Because designing workflows is a notoriously difficult task, it often requires multiple users to collaborate. Not only does the version tree provide an intuitive way to visualize the contribution of different users (e.g., by coloring nodes according to the user who created the corresponding workflow), but the monotonicity of the model allows for simple algorithms for synchronizing changes performed by multiple users.

Provenance information can be easily captured while a workflow is being executed. Once the execution completes, it is also important to maintain *strong* links between a data product and its provenance, i.e., the workflow, parameters and input files used to derive the data product. When data files or provenance are moved or modified, it can be difficult to find the data associated with the provenance or to find the provenance associated with the data. VisTrails provides a persistent storage mechanism that manages input, intermediate, and output data files, strengthening the links between provenance and data. This mechanism provides better support for reproducibility because it ensures the data referenced in provenance information can be readily (and correctly) located. Another important benefit of such management is that it allows caching of intermediate data which can then be shared with other users.

1.3.2 Workflow Execution and Caching

The execution engine in VisTrails was designed to allow the integration of new and existing tools and libraries. We tried to accommodate different styles commonly used for wrapping third-party scientific visualization and computation software. In particular, VisTrails can be integrated with application libraries that exist either as pre-compiled binaries that are executed on a shell and use files as input/outputs, or as C++/Java/Python class libraries that pass internal objects as input/output.

VisTrails adopts a dataflow execution model, where each module performs a computation and the data produced by a module flows through the connections that exist between modules. Modules are executed in a bottom-up fashion; each input is generated on-demand by recursively executing upstream modules (we say module A is *upstream* of B when there is a sequence of connections that goes from A to B). The intermediate data is temporarily stored either in memory (as a Python object) or on disk (wrapped by a Python object that contains information on accessing the data).

To allow users to add their own functionality to VisTrails, we built an extensible package system (see Section 1.3.4). Packages allow users to include their own or third-party modules in VisTrails workflows. A package developer must identify a set of computational modules and for each, identify the input and output ports as well as define the computation. For existing libraries, a compute method needs to specify the translation from input ports to parameters for the existing function and the mapping from result values to output ports.

In exploratory tasks, similar workflows, which share common sub-structures, are often executed in close succession. To improve the efficiency of workflow execution, VisTrails caches intermediate results to minimize recomputation. Because we reuse previous execution results, we implicitly assume that cacheable modules are functional: given the same inputs, modules will produce the same outputs. This requirement imposes definite behavior restrictions on classes, but we believe they are reasonable.

There are, however, obvious situations where this behavior is unattainable. For example, a module that uploads a file to a remote server or saves a file to disk has a significant side effect while its output is relatively unimportant. Other modules might use randomization, and their non-determinism might be desirable; such modules can be flagged as non-cacheable. However, some modules that are not naturally functional can be converted; a function that writes data to two files might be wrapped to output the contents of the files.

1.3.3 Data Serialization and Storage

One of the key components of any system supporting provenance is the serialization and storage of data. VisTrails originally stored data in XML via simple `fromXML` and `toXML` methods embedded in its internal objects (e.g., the version tree, each module). To support the evolution of the schema of these objects, these functions encoded any translation between schema versions as well. As the project progressed, our user base grew, and we decided to support different serializations, including relational stores. In addition, as schema objects evolved, we needed to maintain better infrastructure for common data management concerns like versioning schemas, translating between versions, and supporting entity relationships. To do so, we added a new database (db) layer.

The db layer is composed of three core components: the domain objects, the service logic, and the persistence methods. The domain and persistence components are versioned so that each schema version has its own set of classes. This way, we maintain code to read each version of the schema. There are also classes that define translations for objects from one schema version to those of another. The service classes provide methods to interface with data and deal with detection and translation of schema versions.

Because writing much of this code is tedious and repetitive, we use templates and a meta-schema to define both the object layout (and any in-memory indices) and the serialization code. The meta-schema is written in XML, and is extensible in that serializations other than the default XML and relational mappings VisTrails defines can be added. This is similar to object-relational mappings and frameworks like Hibernate[2] and SQLObject[3], but adds some special routines to automate tasks like re-mapping identifiers and translating objects from one schema version to the next. In addition, we can also use the same meta-schema to generate serialization code for many languages. After originally writing meta-Python, where the domain and persistence code was generated by running Python code with variables obtained from the meta-schema, we have recently migrated to Mako templates[4].

Automatic translation is key for users that need to migrate their data to newer versions of the system. Our design adds hooks to make this translation slightly less painful for developers. Because we maintain a copy of code for each version, the translation code just needs to map one version to another. At the root level, we define a map to identify how any version can be transformed to any other. For distant versions, this usually involves a chain through multiple intermediate versions. Initially, this was a forward-only map, meaning new versions could not be translated to old versions, but reverse mappings have been added for more-recent schema mappings.

Each object has an `update_version` method that takes a different version of an object and returns the current version. By default, it does a recursive translation where each object is upgraded by mapping fields of the old object to those in a new version. This mapping defaults to copying each field to one with the same name, but it is possible to define a method to "override" the default behavior for any field. An override is a method that takes the old object and returns a new version. Because most changes to the schema only affect a small number of fields, the default mappings cover most cases, but the overrides provide a flexible means for defining local changes.

1.3.4 Extensibility Through Packages and Python

The first prototype of VisTrails had a fixed set of modules. It was an ideal environment to develop basic ideas about the VisTrails version tree and the caching of multiple execution runs, but it severely limited long-term utility.

[2] `http://www.hibernate.org`
[3] `http://www.sqlobject.org`
[4] `http://www.makotemplates.org`

We see VisTrails as infrastructure for computational science, and that means, literally, that the system should provide scaffolding for other tools and processes to be developed. An essential requirement of this scenario is extensibility. A typical way to achieve this involves defining a target language and writing an appropriate interpreter. This is appealing because of the intimate control it offers over execution. This appeal is amplified in light of our caching requirements. However, implementing a full-fledged programming language is a large endeavor that has never been our primary goal. More importantly, forcing users who are just trying to use VisTrails to learn an entirely new language was out of the question.

We wanted a system which made it easy for a user to add custom functionality. At the same time, we needed the system to be powerful enough to express fairly complicated pieces of software. As an example, VisTrails supports the VTK visualization library[5]. VTK contains about 1000 classes, which change depending on compilation, configuration, and operating system. Since it seems counterproductive and ultimately hopeless to write different code paths for all these cases, we decided it was necessary to dynamically determine the set of VisTrails modules provided by any given package, and VTK naturally became our model target for a complex package.

Computational science was one of the areas we originally targeted, and at the time we designed the system, Python was becoming popular as "glue code" among these scientists. By specifying the behavior of user-defined VisTrails modules using Python itself, we would all but eliminate a large barrier for adoption. As it turns out, Python offers a nice infrastructure for dynamically-defined classes and reflection. Almost every definition in Python has an equivalent form as a first-class expression. The two important reflection features of Python for our package system are:

- Python classes can be defined dynamically via function calls to the `type` callable. The return value is a representation of a class that can be used in exactly the same way that a typically-defined Python class can.
- Python modules can be imported via function calls to `__import__`, and the resulting value behaves in the same way as the identifier in a standard `import` statement. The path from which these modules come from can also be specified at runtime.

Using Python as our target has a few disadvantages, of course. First of all, this dynamic nature of Python means that while we would like to ensure some things like type safety of VisTrails packages, this is in general not possible. More importantly, some of the requirements for VisTrails modules, notably the ones regarding referential transparency (more on that later) cannot be enforced in Python. Still, we believe that it is worthwhile to restrict the allowed constructs in Python via cultural mechanisms, and with this caveat, Python is an extremely attractive language for software extensibility.

1.3.5 VisTrails Packages and Bundles

A VisTrails package encapsulates a set of modules. Its most common representation in disk is the same representation as a Python package (in a possibly unfortunate naming clash). A Python package consists of a set of Python files which define Python values such as functions and classes. A VisTrails package is a Python package that respects a particular interface. It has files that define specific functions and variables. In its simplest form, a VisTrails package should be a directory containing two files: `__init__.py` and `init.py`.

The first file `__init__.py` is a requirement of Python packages, and should only contain a few definitions which should be constant. Although there is no way to guarantee that this is the

[5] `http://www.vtk.org`

case, VisTrails packages failing to obey this are considered buggy. The values defined in the file include a globally unique identifier for the package which is used to distinguish modules when workflows are serialized, and package versions (package versions become important when handling workflow and package upgrades, see Section 1.4.5). This file can also include functions called `package_dependencies` and `package_requirements`. Since we allow VisTrails modules to subclass from other VisTrails modules beside the root `Module` class, it is conceivable for one VisTrails package to extend the behavior of another, and so one package needs to be initialized before another. These inter-package dependencies are specified by `package_dependencies`. The `package_requirements` function, on the other hand, specifies system-level library requirements which VisTrails, in some cases, can try to automatically satisfy, through its bundle abstraction.

A bundle is a system-level package that VisTrails manages via system-specific tools such as RedHat's RPM or Ubuntu's APT. When these properties are satisfied, VisTrails can determine the package properties by directly importing the Python module and accessing the appropriate variables.

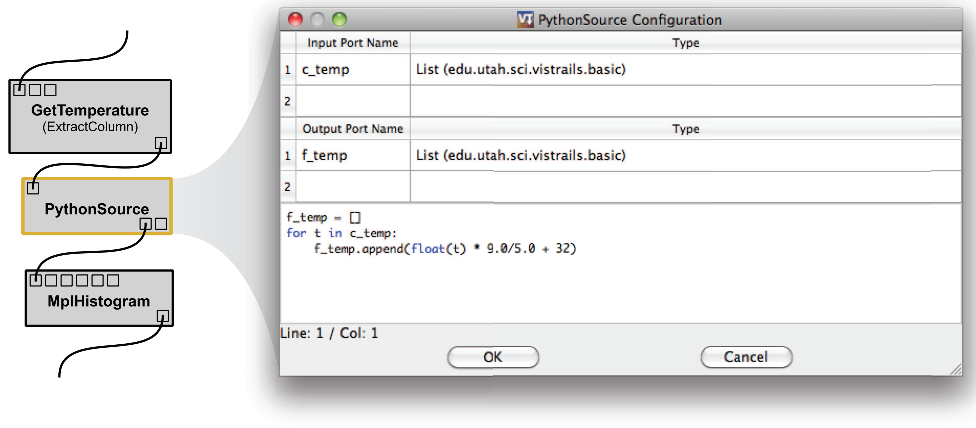
The second file, `init.py`, contains the entry points for all the actual VisTrails module definitions. The most important feature of this file is the definition of two functions, `initialize` and `finalize`. The `initialize` function is called when a package is enabled, after all the dependent packages have themselves been enabled. It performs setup tasks for all of the modules in a package. The `finalize` function, on the other hand, is usually used to release runtime resources (for example, temporary files created by the package can be cleaned up).

Each VisTrails module is represented in a package by one Python class. To register this class in VisTrails, a package developer calls the `add_module` function once for each VisTrails module. These VisTrails modules can be arbitrary Python classes, but they must respect a few requirements. The first of these is that each must be a subclass of a basic Python class defined by VisTrails called, perhaps boringly, `Module`. VisTrails modules can use multiple inheritance, but only one of the classes should be a VisTrails module—no diamond hierarchies in the VisTrails module tree are allowed. Multiple inheritance becomes useful in particular to define class mix-ins: simple behaviors encoded by parent classes which can be composed together to create more complicated behaviors.

The set of available ports determine the interface of a VisTrails module, and so impact not only the display of these modules but also their connectivity to other modules. These ports, then, must be explicitly described to the VisTrails infrastructure. This can be done either by making appropriate calls to `add_input_port` and `add_output_port` during the call to `initialize`, or by specifying the per-class lists `_input_ports` and `_output_ports` for each VisTrails module.

Each module specifies the computation to be performed by overriding the `compute` method. Data is passed between modules through ports, and accessed through the `get_input_from_port` and `set_result` methods. In traditional dataflow environments, execution order is specified on-demand by the data requests. In our case, the execution order is specified by the topological sorting of the workflow modules. Since the caching algorithm requires an acyclic graph, we schedule the execution in reverse topological sorted order, so the calls to these functions do not trigger executions of upstream modules. We made this decision deliberately: it makes it simpler to consider the behavior of each module separately from all the others, which makes our caching strategy simpler and more robust.

As a general guideline, VisTrails modules should refrain from using functions with side-effects during the evaluation of the `compute` method. As discussed in Section 1.3.2, this requirement makes caching of partial workflow runs possible: if a module respects this property, then its behavior is a function of the outputs of upstream modules. Every acyclic subgraph then only needs to be computed once, and the results can be reused.



	Input Port Name	Type
1	c_temp	List (edu.utah.sci.vistrails.basic)
2		

	Output Port Name	Type
1	f_temp	List (edu.utah.sci.vistrails.basic)
2		

```
f_temp = []
for t in c_temp:
    f_temp.append(float(t) * 9.0/5.0 + 32)
```

Figure 1.5: The `PythonSource` module allows users to protoype new functionality

1.3.6 Passing Data as Modules

One peculiar feature of VisTrails modules and their communication is that the data that is passed between VisTrails modules are themselves VisTrails modules. In VisTrails, there is a single hierarchy for module and data classes. For example, a module can provide *itself* as an output of a computation (and, in fact, every module provides a default "self" output port). The main disadvantage is the loss of conceptual separation between computation and data that is sometimes seen in dataflow-based architectures. There are, however, two big advantages. The first is that this closely mimics the object type systems of Java and C++, and the choice was not accidental: it was very important for us to support automatic wrapping of large class libraries such as VTK. These libraries allow objects to produce other objects as computational results, making a wrapping that distinguishes between computation and data more complicated.

The second advantage this decision brings is that defining constant values and user-settable parameters in workflows becomes easier and more uniformly integrated with the rest of the system. Consider, for example, a workflow that loads a file from a location on the Web specified by a constant. This is currently specified by a GUI in which the URL can be specified as a parameter (see the Parameter Edits area in Figure 1.1). A natural modification of this workflow is to use it to fetch a URL that is *computed* somewhere upstream. We would like the rest of the workflow to change as little as possible. By assuming modules can output themselves, we can simply connect a string with the right value to the port corresponding to the parameter. Since the output of a constant evaluates to itself, the behavior is exactly the same as if the value had actually been specified as a constant.

There are other considerations involved in designing constants. Each constant type has a different ideal GUI interface for specifying values. For example, in VisTrails, a file constant module provides a file chooser dialog; a Boolean value is specified by a checkbox; a color value has a color picker native to each operating system. To achieve this generality, a developer must subclass a custom constant from the `Constant` base class and provide overrides which define an appropriate GUI widget and a string representation (so that arbitrary constants can be serialized to disk).

We note that, for simple prototyping tasks, VisTrails provides a built-in `PythonSource` module. A PythonSource module can be used to directly insert scripts into a workflow. The configuration window for PythonSource (see Figure 1.5) allows multiple input and output ports to be specified along with the Python code that is to be executed.

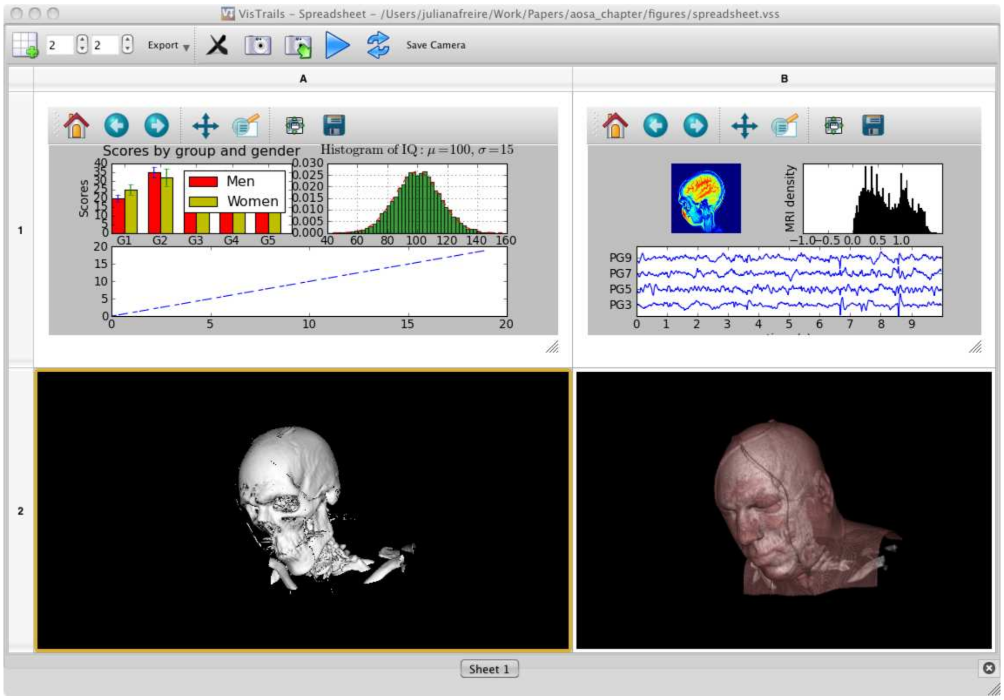

VisTrails - Spreadsheet - /Users/julianafreire/Work/Papers/aosa_chapter/figures/spreadsheet.vss
Export
Save Camera
A
B
1
2
Scores by group and gender
Scores
Men
Women
G1 G2 G3 G4 G5
Histogram of IQ : $\mu=100$, $\sigma=15$
MRI density
PG9
PG7
PG5
PG3
Sheet 1

Figure 1.6: The Visual Spreadsheet presents mutliple workflow results which can be compared side by side. Manipulation widgets (show on the top) can be defined for different types of spreadsheet cells.

1.4 Components and Features

As discussed above, VisTrails provides a set of functionalities and user interfaces that simplify the creation and execution of exploratory computational tasks. Below, we describe some of these. We also briefly discuss how VisTrails is being used as the basis for an infrastructure that supports the creation of provenance-rich publications. For a more comprehensive description of VisTrails and its features, we refer the reader to the VisTrails online documentation.[6]

1.4.1 Visual Spreadsheet

VisTrails allows users to explore and compare results from multiple workflows using the Visual Spreadsheet (see Figure 1.6). The spreadsheet is a VisTrails package with its own interface composed of sheets and cells. Each sheet contains a set of cells and has a customizable layout. A cell contains the visual representation of a result produced by a workflow, and can be customized to display diverse types of data.

To display a cell on the spreadsheet, a workflow must contain a module that is derived from the base `SpreadsheetCell` module. Each `SpreadsheetCell` module corresponds to a cell in the spreadsheet, so one workflow can generate multiple cells. The `compute` method of the `SpreadsheetCell` module handles the communication between the Execution Engine (Figure 1.3) and the spreadsheet. During execution, the spreadsheet creates a cell according to its type on-demand by taking advantage of Python's dynamic class instantiation. Thus, custom visual representations can be achieved by

[6] `http://www.vistrails.org/usersguide`

creating a subclass of `SpreadsheetCell` and having its `compute` method send a custom cell type to the spreadsheet. For example, the workflow in Figure 1.1, `MplFigureCell` is a `SpreadsheetCell` module designed to display images created by matplotlib.

Since the spreadsheet uses PyQt as its GUI backend, custom cell widgets must be subclassed from PyQt's `QWidget`. They must also define the `updateContents` method, which is invoked by the spreadsheet to update the widget when new data arrives. Each cell widget may optionally define a custom toolbar by implementing the `toolbar` method; it will be displayed in the spreadsheet toolbar area when the cell is selected.

Figure 1.6 shows the spreadsheet when a VTK cell is selected, in this case, the toolbar provides specific widgets to export PDF images, save camera positions back to the workflow, and create animations. The spreadsheet package defines a customizable `QCellWidget`, which provides common features such as history replay (animation) and multi-touch events forwarding. This can be used in place of `QWidget` for faster development of new cell types.

Even though the spreadsheet only accepts PyQt widgets as cell types, it is possible to integrate widgets written with other GUI toolkits. To do so, the widget must export its elements to the native platform, and PyQt can then be used to grab it. We use this approach for the `VTKCell` widget because the actual widget is written in C++. At run-time, the `VTKCell` grabs the window id, a Win32, X11, or Cocoa/Carbon handle depending on the system, and maps it to the spreadsheet canvas.

Like cells, sheets may also be customized. By default, each sheet lives in a tabbed view and has a tabular layout. However, any sheet can be undocked from the spreadsheet window, allowing multiple sheets to be visible at once. It is also possible to create a different sheet layout by subclassing the `StandardWidgetSheet`, also a PyQt widget. The `StandardWidgetSheet` manages cell layouts as well as interactions with the spreadsheet in editing mode. In editing mode, users can manipulate the cell layout and perform advanced actions on the cells, rather than interacting with cell contents. Such actions include applying analogies (see Section 1.4.2) and creating new workflow versions from parameter explorations.

1.4.2 Visual Differences and Analogies

As we designed VisTrails, we wanted to enable the *use* of provenance information in addition to its capture. First, we wanted users to see the exact differences between versions, but we then realized that a more helpful feature was being able to apply these differences to other workflows. Both of these tasks are possible because VisTrails tracks the evolution of workflows.

Because the version tree captures all of the changes and we can invert each action, we can find a complete sequence of actions that transform one version to another. Note that some changes will cancel each other out, making it possible to compress this sequence. For example, the addition of a module that was later deleted need not be examined when computing the difference. Finally, we have some heuristics to further simplify the sequence: when the same module occurs in both workflows but was added through separate actions, we we cancel the adds and deletes.

From the set of changes, we can create a visual representation that shows similar and different modules, connections, and parameters. This is illustrated in Figure 1.4. Modules and connections that appear in both workflows are colored gray, and those appearing in only one are colored according to the workflow they appear in. Matching modules with different parameters are shaded a lighter gray and a user can inspect the parameter differences for a specific module in a table that shows the values in each workflow.

The analogy operation allows users to take these differences and apply them to other workflows. If a user has made a set of changes to an existing workflow (e.g., changing the resolution and file format of an output image), he can apply the same changes to other workflows via an analogy. To do so, the user selects a source and a target workflow, which delimits the set of desired changes, as well as the workflow they wish to apply the analogy to. VisTrails computes the difference between the first two workflows as a template, and then determines how to remap this difference in order to apply it to the third workflow. Because it is possible to apply differences to workflows that do not exactly match the starting workflow, we need a soft matching that allows correspondences between similar modules. With this matching, we can remap the difference so the sequence of changes can be applied to the selected workflow [4]. The method is not foolproof and may generate new workflows that are not exactly what was desired. In such cases, a user may try to fix any introduced mistakes, or go back to the previous version and apply the changes manually.

To compute the soft matching used in analogies, we want to balance local matches (identical or very similar modules) with the overall workflow structure. Note that the computation of even the identical matching is inefficient due to the hardness of subgraph isomorphism, so we need to employ a heuristic. In short, if two somewhat-similar modules in the two workflows share similar neighbors, we might conclude that these two modules function similarly and should be matched as well. More formally, we construct a product graph where each node is a possible pairing of modules in the original workflows and an edge denotes shared connections. Then, we run steps diffusing the scores at each node across the edges to neighboring nodes. This is a Markov process similar to Google's PageRank, and will eventually converge leaving a set of scores that now includes some global information. From these scores, we can determine the best matching, using a threshold to leave very dissimilar modules unpaired.

1.4.3 Querying Provenance

The provenance captured by VisTrails includes a set of workflows, each with its own structure, metadata, and execution logs. It is important that users can access and explore these data. VisTrails provides both text-based and visual (WYSIWYG) query interfaces. For information like tags, annotations, and dates, a user can use keyword search with optional markup. For example, look for all workflows with the keyword `plot` that were created by `user: dakoop`. However, queries for specific subgraphs of a workflow are more easily represented through a visual, query-by-example interface, where users can either build the query from scratch or copy and modify an existing piece of a pipeline.

In designing this query-by-example interface, we kept most of the code from the existing Workflow Editor, with a few changes to parameter construction. For parameters, it is often useful to search for ranges or keywords rather than exact values. Thus, we added modifiers to the parameter value fields; when a user adds or edits a parameter value, they may choose to select one of these modifiers which default to exact matches. In addition to visual query construction, query results are shown visually. Matching versions are highlighted in the version tree, and any selected workflow is displayed with the matching portion highlighted. The user can exit query results mode by initiating another query or clicking a reset button.

1.4.4 Persistent Data

VisTrails saves the provenance of how results were derived and the specification of each step. However, reproducing a workflow run can be difficult if the data needed by the workflow is no longer available. In addition, for long-running workflows, it may be useful to store intermediate data as a persistent cache across sessions in order to avoid recomputation.

Many workflow systems store filesystem paths to data as provenance, but this approach is problematic. A user might rename a file, move the workflow to another system without copying the data, or change the data contents. In any of these cases, storing the path as provenance is not sufficient. Hashing the data and storing the hash as provenance helps to determine whether the data might have changed, but does not help one locate the data if it exists. To solve this problem, we created the Persistence Package, a VisTrails package that uses version control infrastructure to store data that can be referenced from provenance. Currently we use Git to manage the data, although other systems could easily be employed.

We use universally unique identifiers (UUIDs) to identify data, and commit hashes from git to reference versions. If the data changes from one execution to another, a new version is checked in to the repository. Thus, the `(uuid, version)` tuple is a compound identifier to retrieve the data in any state. In addition, we store the hash of the data as well as the signature of the upstream portion of the workflow that generated it (if it is not an input). This allows one to link data that might be identified differently as well as reuse data when the same computation is run again.

The main concern when designing this package was the way users were able to select and retrieve their data. Also, we wished to keep all data in the same repository, regardless of whether it is used as input, output, or intermediate data (an output of one workflow might be used as the input of another). There are two main modes a user might employ to identify data: choosing to create a new reference or using an existing one. Note that after the first execution, a new reference will become an existing one as it has been persisted during execution; a user may later choose to create another reference if they wish but this is a rare case. Because a user often wishes to always use the latest version of data, a reference identified without a specific version will default to the latest version.

Recall that before executing a module, we recursively update all of its inputs. A persistent data module will not update its inputs if the upstream computations have already been run. To determine this, we check the signature of the upstream subworkflow against the persistent repository and retrieve the precomputed data if the signature exists. In addition, we record the data identifiers and versions as provenance so that a specific execution can be reproduced.

1.4.5 Upgrades

With provenance at the core of VisTrails, the ability to upgrade old workflows so they will run with new versions of packages is a key concern. Because packages can be created by third-parties, we need both the infrastructure for upgrading workflows as well as the hooks for package developers to specify the upgrade paths. The core action involved in workflow upgrades is the replacement of one module with a new version. Note that this action is complicated because we must replace all of the connections and parameters from the old module. In addition, upgrades may need to reconfigure, reassign, or rename these parameters or connections for a module, e.g., when the module interface changes.

Each package (together with its associated modules) is tagged by a version, and if that version changes, we assume that the modules in that package may have changed. Note that some, or even most, may not have changed, but without doing our own code analysis, we cannot check this. We, however, attempt to automatically upgrade any module whose interface has not changed. To do this, we try replacing the module with the new version and throw an exception if it does not work. When developers have changed the interface of a module or renamed a module, we allow them to specify these changes explicitly. To make this more manageable, we have created a `remap_module` method that allows developers to define only the places where the default upgrade behavior needs to be modified. For example, a developer that renamed an input port 'file' to 'value' can specify that specific remapping so when the new module is created, any connections to 'file' in the old module will now connect to 'value'. Here is an example of an upgrade path for a built-in VisTrails module:

```
def handle_module_upgrade_request(controller, module_id, pipeline):
   module_remap = {'GetItemsFromDirectory':
                       [(None, '1.6', 'Directory',
                         {'dst_port_remap':
                              {'dir': 'value'},
                          'src_port_remap':
                              {'itemlist': 'itemList'},
                          })],
                   }
  return UpgradeWorkflowHandler.remap_module(controller, module_id, pipeline,
                                             module_remap)
```

This piece of code upgrades workflows that use the old `GetItemsFromDirectory` (any version up to 1.6) module to use the `Directory` module instead. It maps the `dir` port from the old module to `value` and the `itemlist` port to `itemList`.

Any upgrade creates a new version in the version tree so that executions before and after upgrades can be differentiated and compared. It is possible that the upgrades change the execution of the workflow (e.g., if a bug is fixed by a package developer), and we need to track this as provenance information. Note that in older vistrails, it may be necessary to upgrade every version in the tree. In order to reduce clutter, we only upgrade versions that a user has navigated to. In addition, we provide a preference that allows a user to delay the persistence of any upgrade until the workflow is modified or executed; if a user just views that version, there is no need to persist the upgrade.

1.4.6 Sharing and Publishing Provenance-Rich Results

While reproducibility is the cornerstone of the scientific method, current publications that describe computational experiments often fail to provide enough information to enable the results to be repeated or generalized. Recently, there has been a renewed interest in the publication of reproducible results. A major roadblock to the more widespread adoption of this practice is the fact that it is hard to create a bundle that includes all of the components (e.g., data, code, parameter settings) needed to reproduce a result as well as verify that result.

By capturing detailed provenance, and through many of the features described above, VisTrails simplifies this process for computational experiments that are carried out within the system. However, mechanisms are needed to both link documents to and share the provenance information.

We have developed VisTrails packages that enable results present in papers to be linked to their provenance, like a deep caption. Using the LaTeX package we developed, users can include figures that link to VisTrails workflows. The following LaTeX code will generate a figure that contains a workflow result:

```
\begin{figure}[t]
\centering{
\vistrail[wfid=119,buildalways=false]{width=0.9\linewidth}
}
\caption{Visualizing a binary star system simulation. This is an image
  that was generated by embedding a workflow directly in the text.
\label{fig:astrophysics}
\end{figure}
```

When the document is compiled using pdflatex, the `\vistrail` command will invoke a Python script with the parameters received, which sends an XML-RPC message to a VisTrails server to execute the workflow with `id 119`. This same Python script downloads the results of the workflow from the server and includes them in the resulting PDF document by generating hyperlinked LaTeX `\includegraphics` commands using the specified layout options (`width=0.9\linewidth`).

It is also possible to include VisTrails results into Web pages, wikis, Word documents and PowerPoint presentations. The linking between Microsoft PowerPoint and VisTrails was done through the Component Object Model (COM) and Object Linking and Embedding (OLE) interface. In order for an object to interact with PowerPoint, at least the `IOleObject`, `IDataObject` and `IPersistStorage` interface of COM must be implemented. As we use the `QAxAggregated` class of Qt, which is an abstraction for implementing COM interfaces, to build our OLE object, both `IDataObject` and `IPersistStorage` are automatically handled by Qt. Thus, we only need to implement the `IOleObject` interface. The most important call in this interface is `DoVerb`. It lets VisTrails react to certain actions from PowerPoint, such as object activation. In our implementation, when the VisTrails object is activated, we load the VisTrails application and allow users to open, interact with and select a pipeline that they want to insert. After they close VisTrails, the pipeline result will be shown in PowerPoint. Pipeline information is also stored with the OLE object.

To enable users to freely share their results together with the associated provenance, we have created crowdLabs.[7] crowdLabs is a social Web site that integrates a set of usable tools and a scalable infrastructure to provide an environment for scientists to collaboratively analyze and visualize data. crowdLabs is tightly integrated with VisTrails. If a user wants to share any results derived in VisTrails, she can connect to the crowdLabs server directly from VisTrails to upload the information. Once the information is uploaded, users can interact with and execute the workflows through a Web browser—these workflows are executed by a VisTrails server that powers crowdLabs. For more details on how VisTrails is used to created reproducible publications, see `http://www.vistrails.org`.

1.5 Lessons Learned

Luckily, back in 2004 when we started thinking about building a data exploration and visualization system that supported provenance, we never envisioned how challenging it would be, or how long it would take to get to the point we are at now. If we had, we probably would never have started.

Early on, one strategy that worked well was quickly prototyping new features and showing them to a select set of users. The initial feedback and the encouragement we received from these users was instrumental in driving the project forward. It would have been impossible to design VisTrails without user feedback. If there is one aspect of the project that we would like to highlight is that most features in the system were designed as direct response to user feedback. However, it is worthy to note that many times what a user asks for is not the best solution for his/her need—being responsive to users does not necessarily mean doing exactly what they ask for. Time and again, we have had to design and re-design features to make sure they would be useful and properly integrated in the system.

Given our user-centric approach, one might expect that every feature we have developed would be heavily used. Unfortunately this has not been the case. Sometimes the reason for this is that the feature is highly "unusual", since it is not found in other tools. For instance, analogies and even the version tree are not concepts that most users are familiar with, and it takes a while for them to get comfortable with them. Another important issue is documentation, or lack thereof. As with many other open source projects, we have been much better at developing new features than at documenting the existing ones. This lag in documentation leads not only to the underutilization of useful features, but also to many questions on our mailing lists.

[7]`http://www.crowdlabs.org`

One of the challenges of using a system like VisTrails is that it is very general. Despite our best efforts to improve usability, VisTrails is a complex tool and requires a steep learning curve for some users. We believe that over time, with improved documentation, further refinements to the system, and more application- and domain-specific examples, the adoption bar for any given field will get lower. Also, as the concept of provenance becomes more widespread, it will be easier for users to understand the philosophy that we have adopted in developing VisTrails.

1.5.1 Acknowledgments

We would like to thank all the talented developers that contributed to VisTrails: Erik Anderson, Louis Bavoil, Clifton Brooks, Jason Callahan, Steve Callahan, Lorena Carlo, Lauro Lins, Tommy Ellkvist, Phillip Mates, Daniel Rees, and Nathan Smith. Special thanks to Antonio Baptista who was instrumental in helping us develop the vision for the project; and Matthias Troyer, whose collaboration has helped us to improve the system, and in particular has provided much of the impetus for the development and release of the provenance-rich publication functionality. The research and development of the VisTrails system has been funded by the National Science Foundation under grants IIS 1050422, IIS-0905385, IIS 0844572, ATM-0835821, IIS-0844546, IIS-0746500, 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.

Bibliography

[1] Louis Bavoil, Steve Callahan, Patricia Crossno, Juliana Freire, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. VisTrails: Enabling interactive multiple-view visualizations. In *Proceedings of IEEE Visualization*, pages 135–142, 2005.

[2] Juliana Freire, David Koop, Emanuele Santos, and Cláudio T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engineering*, 10(3):11–21, 2008.

[3] Juliana Freire, Cláudio T. Silva, Steve Callahan, Emanuele Santos, Carlos E. Scheidegger, and Huy T. Vo. Managing rapidly-evolving scientific workflows. In *International Provenance and Annotation Workshop (IPAW)*, LNCS 4145, pages 10–18. Springer Verlag, 2006.

[4] Carlos E. Scheidegger, Huy T. Vo, David Koop, Juliana Freire, and Cláudio T. Silva. Querying and creating visualizations by analogy. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1560–1567, 2007.