The Provenance of Workflow Upgrades

David Koop[1], Carlos E. Scheidegger[2], Juliana Freire[1], and Cláudio T. Silva[1]

[1] University of Utah
[2] AT&T Research

Abstract

Provenance has become an increasingly important part of documenting, verifying, and reproducing scientific research, but as users seek to extend or share results, it may be impractical to start from the exact original steps due to system configuration differences, library updates, or new algorithms. Although there have been several approaches for capturing workflow provenance, the problem of managing upgrades of the underlying tools and libraries orchestrated by workflows has been largely overlooked. In this paper we consider the problem of maintaining and re-using the provenance of workflow upgrades. We propose different kinds of upgrades that can be applied, including automatic mechanisms, developer-specified, and user-defined. We show how to capture provenance from such upgrades and suggest how this provenance might be used to influence future upgrades. We also describe our implementation of these upgrade techniques.

1 Introduction

As tools that capture and utilize provenance are accepted by the scientific community, they must provide capabilities for supporting reproducibility as systems evolve. Like any information stored or archived, it is important that provenance is usable both for reproducing prior work and migrating that work to new environments. Just as word processing applications allow users to load old versions of documents and convert them to newer versions and data processing libraries provide migration paths for older formats, provenance-enabled tools should provide paths to upgrade information to match newer software or systems. Furthermore, it is important to capture and understand the changes that were made in order to run a previous computation in a new environment. One goal in documenting provenance is that users can more easily verify and extend existing work. If a given computation cannot be translated to newer systems or software versions, extensions become more difficult.

Workflow systems have made significant strides in allowing users to quickly compose a variety of tools while automatically capturing provenance information during workflow creation and execution [11, 8]. Such systems enforce a structure on computations so that each workflow step is easily identifiable. Unfortunately, while these systems provide interfaces to a variety of routines and libraries, they are limited in their ability to upgrade workflows when the underlying routines or their interfaces are updated. It is well-known that software tends to *age* [19].

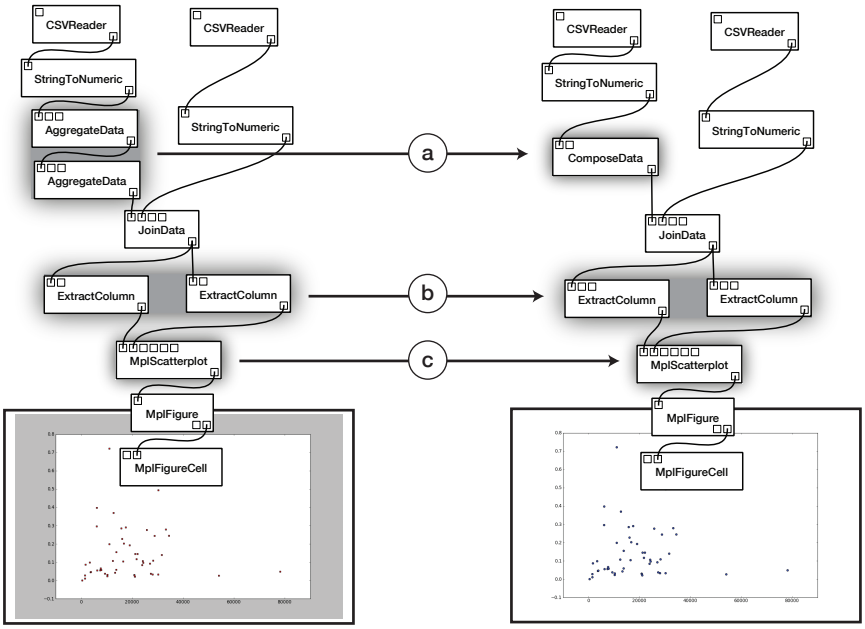
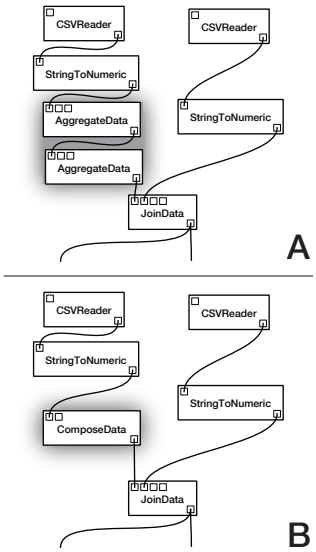


Fig. 1. A workflow comparing road maintenance and number of miles of road by state before and after upgrading two packages. In (a), the `AggregateData` module has been replaced, and the developer has specified an upgrade to combine multiple aggregation steps into a single `ComposeData` module. In (b), the interface of `ExtractColumn` has been updated to offer a new parameter. Finally, in (c), the interface of the plotting mechanism has not changed, but the implementation of that module has, as evidenced by the difference in the background of the resulting plots.

As requirements change, so do implementations and interfaces. This is more starkly obvious in the case of workflows, where different software tools from a variety of different sources need to be orchestrated. Figure 1 shows an example of different modifications that can be applied to workflow modules, including the addition of new parameters, the merger of two modules, and the replacement of the underlying computation. Still, many workflow systems do store information about the versions of routines as provenance. We seek to use this information to design schemes that allow users to migrate their work as newer algorithms and systems are developed.

There are two major approaches when dealing with upgraded software components and the documents or applications that utilize them. It is often important to maintain old versions of libraries and routines for existing applications that rely on them. In this case, an upgrade to a library should not replace the existing version but rather augment existing versions. Such an approach is common in system libraries and Web services where deleting previous versions can render existing code unexecutable. However, when we can safely upgrade the document or application to match the new interfaces, we might modify the object to utilize the new version. This second approach is more often used for documents than



Upgrade Provenance (A → B)
delete connection `StringToNumeric` → `AggregateData`
delete connection `AggregateData` → `AggregateData`
delete module `AggregateData` version 1.0.4
delete connection `AggregateData` → `JoinData`
delete module `AggregateData` version 1.0.4
add module `ComposeData` version 1.1.0
add connection `StringToNumeric` → `ComposeData`
add connection `ComposeData` → `JoinData`

Execution Provenance (A)
execute module `CSVReader` version 0.8.0
execute module `StringToNumeric` version 0.9.0
→ **execute module** `AggregateData` **version 1.0.4**
→ **execute module** `AggregateData` **version 1.0.4**
execute module `CSVReader` version 0.8.0
execute module `StringToNumeric` version 0.9.0
execute module `JoinData` version 1.0.0
...

Execution Provenance (B)
execute module `CSVReader` version 0.8.0
execute module `StringToNumeric` version 0.9.0
→ **execute module** `ComposeData` **version 1.1.0**
execute module `CSVReader` version 0.8.0
execute module `StringToNumeric` version 0.9.0
execute module `JoinData` version 1.0.0
...

Fig. 2. On the right, we show the provenance of upgrading workflow (A) to the updated workflow (B). Besides the provenance of the upgrade, here we show the provenance of the executions of both (A) and (B). Note that version information is maintained in both forms of provenance.

for existing applications or code, because there exists an application that can upgrade old versions. While the first approach is important to to ensure that the original work can be replicated, because workflows are only loosely coupled to their implementations and live in the context of a workflow system, this second approach is sensible for them. Furthermore, as Figure 2 illustrates, by capturing the provenance of the upgrades, we know exactly what has been changed from the original version and how it might be reverted.

In order to accomplish the goal of upgrading an existing workflow, we must solve the challenges of detecting when upgrades are necessary and applicable, as well as dealing with routines (modules) from disparate sources. Because workflow systems often store information about the modules included in workflows, it is possible to detect when the current implementation of a module differs from one that was previously used. However, since they come from different sources, each source may define or release upgrades differently. Thus, we cannot hope to upgrade workflows atomically, without considering specific concerns from each source. Finally, while some upgrades may be automated or specified by a developer, others may require user intervention. When the user needs to be in the loop, it is important to make the process less tedious and error-prone.

Contributions. We propose a routine for detecting when a workflow is incompatible with current installed software and approaches for both automated and user-defined upgrades. Our automated algorithm combines default routines for

cases when only implementations changes with developer-defined routines, and uses a piecewise algorithm to process all components from each package at once. This allows complex upgrades, like replacing a subworkflow containing three modules with a single module. For user-defined upgrades, we suggest how a user might define a single upgrade once and apply it automatically to a collection of workflows. Finally, we discuss how upgrades should be considered as an integral part of the information currently managed by provenance-enabled systems. It is critical that we can determine what steps may have led to an upgraded workflow producing different results from the original. We describe our implementation of this upgrade framework in the VisTrails system [26].

2 Workflow Upgrades

2.1 Background

A *workflow* describes a set of computations as well as an order for these computations. To simplify the presentation, we focus on dataflows; but note that our approach is applicable to more general workflow models. In a dataflow, computational flow is dictated by the data requirements of each computation. A dataflow is represented as a directed acyclic graph where nodes are the computational *modules* and edges denote the data dependencies as *connections* between the modules—an edge connects the *output port* of a module to an *input port* of another. Often, a module has a set of associated *parameters* that can control the specifics of one computation. Some workflows also utilize *subworkflows* where a single module is itself implemented by an underlying workflow.

Because workflows abstract computation, there must be an association between the module instances in a workflow and the underlying execution environment. This link is managed by the *module registry* which maps module identifiers to their implementations. For convenience and maintenance, related modules are often grouped together in *packages*. Thus, the module identifier may consist of package identifier, a module name, an optional namespace, and information about the version of the implementation. Version information can serve to inform us when implementations or interfaces in the environment change.

Consider, for example, the VisTrails system [26]. In VisTrails, each module corresponds to a Python class that derives from a pre-defined base class. Users define custom behaviors by implementing a small set of methods. These, in turn, might run some code in a third-party library or invoke a remote procedure call via a Web service. The Python class also explicitly describes the interface of the module: the set of allowed input and output connections, given by the module's *ports*. A VisTrails package consists of a set of Python classes.

Incompatible Workflows. After a workflow is created, changes to the underlying implementation of one or more of its modules may make the workflow *incompatible*. Figure 3 shows an incompatible and a valid version of a workflow. Because module registry information is usually not serialized with each workflow, it can be difficult for users to define upgrades for obsolete workflows. As shown in the

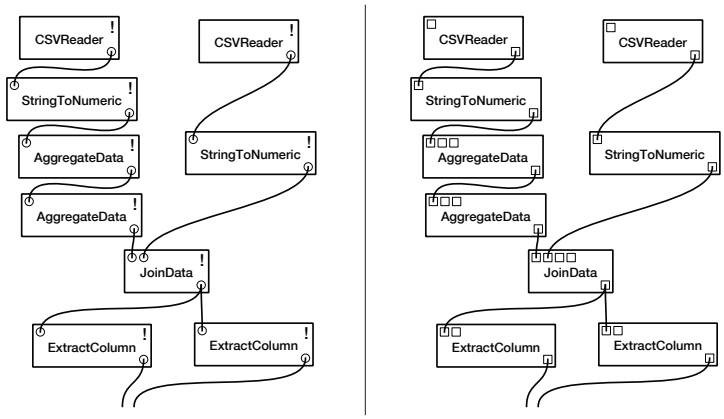


Fig. 3. Incompatible (left) and valid (right) versions of a workflow. In an incompatible workflow, the implementation of modules is missing, and thus, no information is available about the input and output ports of these modules.

figure, although we may lack the appropriate code to execute a module or display the complete set of input and output interfaces for a module, we can display each module with the subset of ports identified by connections in the workflow. This is useful to allow users to edit incompatible workflows in order to make them compatible with their current environment.

Provenance of Module Implementation. Workflow systems offer mechanisms for capturing provenance information both about the evolution of the workflow itself and each execution of a workflow [11, 12, 21]. Information about the implementations used for each workflow module may be stored together with either evolution or execution provenance (i.e., the execution log). However, note that if the interface for a module changes, it will often require a change in the workflow specification. Thus, while the execution provenance may contain information about the versions used to achieve a result, any change in the interface of a module may make reproducibility via execution provenance alone difficult. By storing information about the implementations (like versions of each module) as evolution provenance, we can connect the original workflow to all upgraded versions.

Workflow evolution can be captured via *change-based provenance* [12], where every modification applied to a workflow is recorded. The set of changes is represented as a tree where nodes correspond to workflow versions and an edge between two nodes corresponds to the difference between the two corresponding workflow specifications.

Any workflow instance can be reconstructed by applying the entire sequence of change operations from the root node to the current version. For upgrades, we can leverage this approach to record the set of changes necessary to update an old workflow to a new version. Note that these changes define the difference

between the two versions, so our provenance will maintain an explicit definition of the upgrade for reference and comparison.

2.2 Detecting the Need for Upgrades

To support upgrades, workflow systems must provide developers with facilities to develop and maintain different versions of modules (and packages) as well as detect and process inconsistencies when workflows created with older versions of modules are materialized. First, it is important to have a mechanism for identifying a group of modules (e.g., using a group key), as well as a version indicator or some other method like content-hashing that can be used to identify when module implementations may have changed. Ideally, any version identifier of a module should reflect the version of the code or underlying libraries. In fact, we may be able to aid developers by signaling when their code has changed, alerting them to the need to change the version. Alternately, developers might link version identifiers to revisions of their code as defined in a version control system.

Second, we need to tackle the problem of identifying when and where upgrades might be necessary. Upon opening a workflow, the system needs to check that the modules specified are consistent with the implementation defined by the module registry. As discussed earlier, this usually involves checking version identifiers but could also be based on actual code. If there are inconsistencies, we need to identify the type of discrepancy; the workflow may specify an obsolete version of a module, a newer version, or perhaps the module may not exist in the current registry. In all of these cases, we need to reconcile the workflow to the current environment.

2.3 Processing Upgrades

We wish to allow developers to specify upgrade paths but also provide automated routines when upgrades are trivial and allow users to override the specified paths. The package developer can specify how a specific module is to be upgraded in all contexts. If that is not possible or the information is not available for a given module, we can attempt to automatically upgrade a module by replacing the old version with a new version of the same module. A third method for upgrading a workflow is to display the obsolete modules and let the user replace them directly. Our upgrade framework leverages all three approaches. It starts with developer-specified changes, provides default, automated upgrades if the developer has not provided them, and allows the user to choose to accept the upgrade, modify it, or design their own.

Developer-Defined Upgrades. Because the modules of any workflow may originate from a number of different packages, we cannot assume that a global procedure can upgrade the entire workflow. Instead, we allow developers to specify upgrade routines for each package. Specifically, we allow them to write a method which accepts the workflow and the list of incompatible modules. A module may be

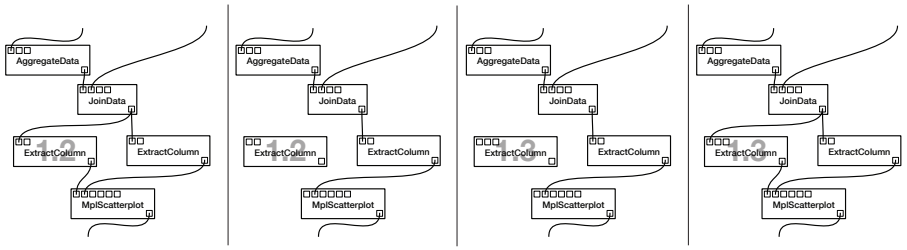


Fig. 4. Upgrading a single module automatically involves deleting all connections, replacing the module with the new version, and finally adding the connections back.

incompatible because it no longer exists in the package or its version is different from the implementation currently in the registry. A developer needs to implement solutions to handle both of these situations, but the system can provide utility routines to minimize the effort necessary for some types of changes. In addition, there may be cases where the developer wants to replace entire subworkflows with different ones. Changes in the specification of parameter values may also require upgrade logic. For example, an old version of module may have taken the color specification as four integers in the [0,255] range, but the new version requires floats in the [0.0, 1.0] range. Such conversions can be developer-specified so that the a user need not modify their workflows in order for them to work with new package versions. Note that developer-specified upgrades may need to be aware of the initial version of a module. For example, if version 0.1 of a module has a certain parameter, version 0.2 removes it, and version 1.0 adds it back, the upgrade from 0.1 to 1.0 will be necessarily different than the upgrade from 0.2 to 1.0.

Automatic Upgrades. We can attempt to automate upgrades by replacing the original module with a new version of the same module. For any module that needs an upgrade, we check the registry for a module that shares the same identifying information (excluding version) and use that module instead. Note that it is necessary to recreate all incoming and outgoing connections because the old module is deleted and a new module is added. If an upgraded module renames or removes a port, it is not possible to complete the upgrade. We can either continue with other upgrades and notify the user, or rollback all changes and alert the user. Also note that if two connected modules both require upgrades we will end up deleting and adding at least one of the connections twice, once for the first module replacement and again for the second module upgrade. Finally, we need to transfer parameters to the new version in a similar procedure as that used with connections. See Figure 4 for an example of an automatic upgrade.

User-Assisted Upgrades. While we hope that automatic and developer-specified upgrades will account for most of the cases, they may fail for complicated situations. In addition, a developer may not specify all upgrade paths or a user may

desire greater control over the changes. In such a scenario, we need to display the old, incompatible pipeline, highlight modules that are out-of-date, and allow the user to perform standard pipeline manipulations. One problem is that, because we may not have access to the version of the package that was used to create the workflow, we may not be able to display the module correctly for the user to interact with. With VisTrails, we can display the basic graph connectivity as shown in Figure 3, but we may not have entire module specification. Our display is therefore a "recovery mode" where the workflow is shown but cannot be executed or interacted with in the same way as a valid version. Once users replace all old modules with current versions, they will able to execute the workflow and interact with it. We can aid users by providing high-level actions that allow them to, for example, replace an incompatible module with a new, valid one.

In addition, while users may be willing to perform one or two upgrades manually, it would be helpful if we are able to aid users by automating future upgrades based on those they have already defined. Workflow analogies provide this functionality by allowing users to select existing actions including upgrades and apply them to other workflows [20]. Because analogies compute a soft matching between starting workflows, they can be applied to a variety of different workflows. Thus, for a large collection of workflows, a user may define a few upgrades and compute the rest automatically using these analogies.

2.4 Provenance Concerns

Given a data product, we cannot hope to reproduce or extend the data product without knowing its provenance—how it was generated. If our provenance information includes information about the versions of the modules used, we can use that to drive upgrades. Note that without version information, we may incorrectly determine which upgrades are necessary. Thus, the provenance of the original workflow is important to define the upgrade.

At the same time, we wish to capture the provenance of the upgrades. When users either run old versions of workflows or upgrade and modify these versions, it is important to track the changes both in the execution provenance and in the workflow evolution provenance. By noting the specific module versions used in the execution provenance, we can better support reproducibility. We need to ensure that the versions recorded are exactly the versions executed, not allowing silent upgrades to happen without being noted in the provenance. Similarly, whenever a user upgrades a workflow, the changes that took place should be noted as evolution provenance so that subsequent changes are captured correctly. See Figure 2 for an example of captured provenance information that is relevant for upgrades.

As a workflow evolves over a number of years and is modified by a number of users, it is important to track the provenance of this evolution. Upgrades may be critical changes in workflow development and often occur when a new user starts to revise an existing result. By keeping track of these actions, we may be able to identify how, for example, inconsistencies in results may have arisen because of an upgrade. In addition, we do not lose links as workflows are refined.

Without upgrades, a user may create a (duplicate) workflow rather than re-use an existing one. If that occurs, we lose important provenance of the original workflow and related workflows.

3 Implementation

We have implemented the framework described in Section 2 in the VisTrails system. Below, we describe this implementation.

In VisTrails, when a workflow is loaded (or materialized), it is validated against the current environment: the classes defining the modules and the port types for each module. To detect whether modules have changed, we begin by checking each module and ensuring the requested version matches the registry version. Next, we check each connection to ensure that the ports they connect are also valid. Finally, we check the parameter types to ensure they match those specified by the implementation. If any mismatches are detected, we raise an exception that indicates what the problem is and which part of the workflow it affects. Note that if one problem occurs, we can immediately quit validation and inform the user, but if we wish to fix the problems, it is useful to identify all issues. Thus, we collect all exceptions during validation, and pass them to a handler.

We attempt to process all upgrades at once, with the exception of subworkflows which are processed recursively. To this end, we sort all requests by the packages that they affect, and attempt to solve all issues one package at a time. This way, a package developer can write a handler to process a group of upgrade requests instead of processing each request individually.

Replace, Remap, and Copy. Note that an upgrade that deletes an old module and adds a new version discards information about existing connections, parameters, and annotations. In order to maintain this information as well as its provenance, we extended VisTrails change-based provenance with a new change type (or action) that replaces the original module, remaps the old information, and copies it to a new version of the module. This ensure that we transfer all relevant information to the new version and maintain its provenance. The new action extracts information about connections, parameters and annotations from the old module before replacing it, and then adds that information to the new module. Note that, because interfaces may change, we allow the user to remap parameter, port, or annotation names to match the new module's interface.

Algorithm. Formally, our algorithm for workflow upgrades takes a list of detected inconsistencies between a workflow and the module registry and produces a set of actions to revise the workflow. We categorize inconsistencies as "missing", "obsolete", or "future" modules, and this information is encoded in the exception allowing developers to tailor upgrade paths accordingly. We begin by sorting these errors by package identifier. Then for each package, we check if the package has a handler for all types of upgrades. If it does, we call that handler.

If not, we cannot hope to reconcile "missing" modules. For obsolete modules, we can attempt to automatically reconcile them by replacing them with newer versions. For future modules, we can attempt to downgrade them automatically, but usually we raise this error to the user.

Automatic upgrades work module by module, and for each module, we first check to see if an upgrade is possible before proceeding. An upgrade is possible if the module interface has not changed from the version specified by the workflow and the version that exists on the system. We check that by seeing if each connection and parameter setting can be trivially remapped. If they can, we extract all of the connections and parameters before deleting all connections to the module and the module itself. Then, we add the new version of the module and replace the connections and parameters. All of these operations are encoded as a single action.

For developer-defined upgrades, we pass all of the information about inconsistencies as well as the current state of the workflow to the package's upgrade handler. The handler can make use of several capabilities of the workflow system to minimize the amount of code. Specifically, we have a remap function that allows a developer to specify how to replace a module when interface changes are due to renaming. In addition, developers can replace entire pieces of a workflow, but doing so might require locating subworkflows that match a given template. Many workflow systems already have query capabilities, and these can be applied to facilitate these more complex upgrades. As with automatic upgrades, these operations are encoded as a single action.

If automated and developer-defined upgrades cannot achieve a compatible workflow, a user can define an upgrade path. Most of this process is manual and mirrors how a user might normally update a workflow. Until the workflow is compatible with the current environment, the workflow cannot be executed, giving users a well-defined goal. Upon achieving a valid workflow, we can save the user's actions and use workflow analogies [20] to help automate future upgrades.

Subworkflows. To handle subworkflows, both validation and upgrade handling are performed recursively. Thus, we process any workflow by first recursing on any subworkflow modules, processing the underlying workflows, and then continuing with the rest of the workflow. However, our upgrades must be handled using an extra step; if we update a subworkflow, we must also update the module tied to that subworkflow to reflect any changes. For example, a subworkflow may modify its external interface by deleting inputs or outputs. Thus, we must upgrade a module after updating its underlying subworkflow.

Preferences. While upgrades are important, we wish to add them without interfering with a user's normal work. Besides the choice between upgrading or trying to load the exact workflow with older package versions, a user may also wish to be notified of upgrades and persist their provenance in different ways. Specifically, if old versions exist, a user may wish to always try use them, automatically upgrade, or be prompted for a decision. If not, a user has a similar selection of options: never upgrade, always upgrade, or be presented with the

choice to upgrade. When a user wants to upgrade, he may choose to persist the provenance of these upgrades immediately or delay saving these changes until other changes occur. If a user is browsing workflows, it may may reasonable to only persist upgrade provenance when the workflow is modified or run. This way, a user can examine a workflow as it would appear after an upgrade, but the persistence of these upgrades is delayed until something is changed or the workflow is executed. Users might also want to have immediate upgrades where the upgrade provenance is persisted exactly when any workflow is upgraded, even if the user is only viewing the workflow.

4 Discussion

While perfect reproducibility cannot be guaranteed without maintaining the exact system configuration and libraries, we believe that workflow upgrades offer a sensible approach to manage the migration from older workflows to new environments. Note that provenance allows us to always revisit the original workflow, and we can run this version if we can reconstruct the same environment. By storing the original implementations along with workflows, we may be able to reproduce the original run, although changes in the system configuration may limit such runs. Thus, coupling provenance with version control systems could ensure that we users can access previous package implementations. However, when extending prior work in new environments, upgrades also serve to convert older work to more efficient and extensive environments. In addition, managing multiple software versions is a non-trivial task, and even with a modern OS package management system, installing a given package in the presence of conflicts is actually known to be NP-Complete [9]. Thus, we cannot expect in general to easily run arbitrarily old library versions. Because workflows abstract the implementation from the computational structure, the results of upgrades are more likely be valid.

Some workflow systems use Web services or other computational modules that are managed externally. In these cases, we may not know if the interface or implementation may have changed so it is harder to know when upgrades are necessary. However, the services may make version information available or the workflow system may be able to detect a change in the interface [5]. In this case, we are not able to leverage developer-specified upgrade routines, but we should be able to accomplish automatic or user-specified upgrades.

When using change-based provenance to track upgrades, a user can see both the original evolution as well as the upgrades and progress after the upgrades. See Figure 5 for an example. It may be useful to upgrade an entire collection of related workflows while retaining the original provenance of exploration, but adding the upgraded versions may lead to a complex interface. We believe that restructuring the tree to display the original history but with links to the upgrades might be useful. Finally, we emphasize that the change-based model for the workflows provenance in VisTrails is an attractive medium in which to incorporate the upgrading data. Since the upgrades as represented as actions, they are

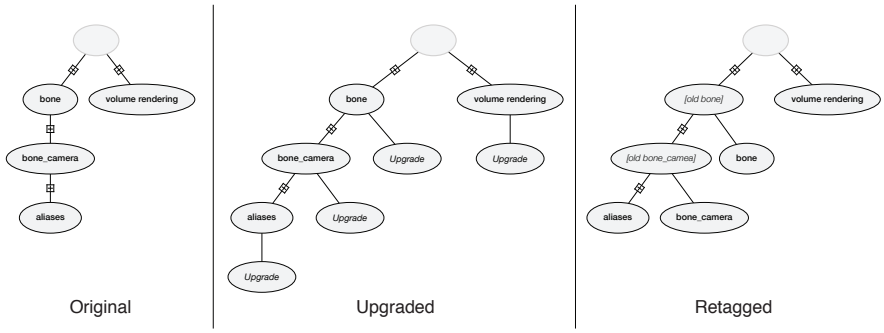


Fig. 5. Workflow Evolution before and after upgrades as well as after retagging the nodes.

treated as first-class data in the system, and so the extensive process provenance capabilities of VisTrails can be directly used. For example, upgrade actions can then be used in queries or incorporated into statistical analyses [14, 20, 21].

5 Related Work

Workflow systems have recently emerged as an attractive alternative for representing and managing complex computational tasks. The goal behind these systems is to provide the utility of the shell script in a more user-friendly, structured manner. Workflow systems incorporate comprehensive metadata which, among other advantages, facilitates programming and distribution of results [15], reproducibility [12], allows better execution monitoring [18], and provides potential efficiency gains [3].

As the auditability and cost of generating results has increased, managing the provenance of data products [22] and computational processes [12] has become very important. Together, these ideas allow users to obtain a fairly comprehensive picture of the programs and data that were used to generate final results. However, these descriptions are, in a sense, static. In general, the processes are assumed to stay the same for the lifetime of the workflow, and, as we have argued before, longevity necessarily introduces changing requirements and interfaces. Our approach serves to detect and manage these changes to underlying implementations while still keeping the attractive features of workflow systems described above.

It is well known that longevity introduces novel challenges for maintainability of software systems, in particular in the presence of complicated dependencies [19]. There have been a number of approaches to problem of managing software upgrades, in particular, in understanding and ensuring safety properties of dynamic updates in, for example, running code or persistent stores [4, 10]. In small-scale environments, the solutions tend to involve the description (or prediction) of desired properties to be maintained [16]. For deployments at

the scale of entire institutions or large computer clusters, they tend to involve careful scheduling, and staged deployment of upgrades [1, 6].

Component-based software has evolved to separate different concerns in order to provide wide-ranging functionality. While usually at a lower level than workflow systems, component objects use well-defined interfaces and are substitutable [24]. For this reason, it is also important to track the component evolution and versioning [23]. The term "dependency hell" was coined to describe problems with compatibility when replacing components with new versions. McCamant and Ernst describe methods to identify such incompatibilities [17] while Stuckenholz proposes "intelligent component swapping" to update multiple components at once [23].

Web services are another kind of component-based architecture. Since the standards do not address the evolution of Web services, developers must rely on design patterns and best practices [5]. Specifically, adding to an interface is possible, but changing or removing from that interface is not. Andrikopoulos et al. formalize the concepts of service evolution [2]. There are a variety of approaches that seek to develop mechanisms to version Web services including using a chain of adapters [13] and hierarchical abstraction [25]. In order to publish such versions, services are distinguished via namespaces or URLs. In contrast to much of the work for component upgrades, our approach seeks to add capability by updating older workflows rather than only maintaining backward compatibility.

In this paper, we focus on the problem of providing a means of describing upgrade paths so that a workflow can be automatically updated, its upgrade history appropriately recorded, and its execution sufficiently similar to the one before the upgrade. Such problems exist even when lower-level upgrades are successfully deployed. In that sense, our mechanisms for coping with upgrades are closer in spirit to mechanisms for automatically updating database queries after relational schemas have changed [7].

6 Conclusion & Future Work

We have proposed a framework for workflow upgrades and described its implementation in the VisTrails system. Our framework handles three types of upgrades—automated, developer-specified, and user-defined, and we have discussed how these can be supported in a systematic fashion. We have also shown how the framework leverages provenance information to accomplish upgrades and produces updated provenance detailing the changes introduced by the upgrades. Our implementation is currently available in nightly releases of VisTrails, and we are planning to incorporate it into the next major release of VisTrails.

One area that we would like to explore further is the interface for involving the user in upgrades. The "replace module" action allows users to specify how an upgrade is accomplished, and we believe a user might drag a new module onto the incompatible module to replace it. At the same time, if the routine specifications do not exactly match, the user should be able to specify the remapping,

similar to the method available to developers in their code. We might extend this functionality to allow the user to specify the connections visually.

In addition to capturing the provenance of upgrades and using this information to guide future user-driven, manual upgrades, we believe we might also use this provenance for further analysis. For example, we might be able to examine the actions used in upgrades to mine rules for packages whose developers have not defined upgrade paths. It may also be interesting to try to analyze performance or accuracy changes in workflow execution after upgrades.

7 Acknowledgments

Our research has been funded by the National Science Foundation (grants IIS-0905385, IIS-0844546, IIS-0746500, ATM-0835821, CNS-0751152, IIS-0713637, OCE-0424602, IIS-0534628, CNS-0514485, IIS-0513692, CNS-0524096), the Department of Energy SciDAC (VACET and SDM centers, and SBIR DE-FG02-85157), and IBM Faculty Awards (2005, 2006, 2007, and 2008).

References

1. S. Ajmani, B. Liskov, and L. Shrira. Scheduling and simulation: how to upgrade distributed systems. In *HOTOS*, pages 8–8, 2003.
2. V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou. Managing the evolution of service specifications. In *CAiSE '08: Proceedings of the 20th international conference on Advanced Information Systems Engineering*, pages 359–374, Berlin, Heidelberg, 2008. Springer-Verlag.
3. L. Bavoil, S. Callahan, P. Crossno, J. Freire, C. Scheidegger, C. Silva, and H. Vo. VisTrails: Enabling interactive, multiple-view visualizations. In *Proceedings of IEEE Visualization*, pages 135–142, 2005.
4. C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, 38(11):403–417, 2003.
5. K. Brown and M. Ellis. Best practices for Web services versioning. *IBM developerWorks*, 2004. http://www.ibm.com/developerworks/webservices/library/ws-version/.
6. O. Crameri, N. Knezevic, D. Kostic, R. Bianchini, and W. Zwaenepoel. Staged deployment in mirage, an integrated software upgrade testing and distribution system. *SIGOPS Oper. Syst. Rev.*, 41(6):221–236, 2007.
7. C. Curino, H. J. Moon, and C. Zaniolo. Automating database schema evolution in information system upgrades. In *HotSWUp*, pages 1–5, 2009.
8. S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of SIGMOD*, pages 1345–1350, 2008.
9. R. di Cosmo. Report on formal management of software dependencies. Technical report, INRIA, Sep 2005. EDOS Project Deliverable WP2-D2.1.
10. T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system. In *Middleware*, pages 1–20, 2009.
11. J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engineering*, 10(3):11–21, 2008.

12. J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo. Managing rapidly-evolving scientific workflows. In *IPAW*, pages 10–18, 2006. Invited paper.
13. P. Kaminski, M. Litoiu, and H. Müller. A design technique for evolving web services. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 23, New York, NY, USA, 2006. ACM.
14. L. Lins, D. Koop, E. Anderson, S. Callahan, E. Santos, C. Scheidegger, J. Freire, and C. Silva. Examining statistics of workflow evolution provenance: a first study. In *Proceedings of SSDBM*, 2008.
15. B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. A. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the kepler system: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(10):1039–1065, 2006.
16. S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC*, pages 287–296, 2003.
17. S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP 2004 — Object-Oriented Programming, 18th European Conference*, pages 440–464, Oslo, Norway, June 16–18, 2004.
18. Microsoft Workflow Foundation. http://msdn2.microsoft.com/en-us/netframework/aa663328.aspx.
19. D. L. Parnas. Software aging. In *ICSE*, pages 279–287, 1994.
20. C. Scheidegger, H. Vo, D. Koop, J. Freire, and C. Silva. Querying and creating visualizations by analogy. *IEEE TVCG*, 13(6):1560–1567, 2007.
21. C. E. Scheidegger, D. Koop, E. Santos, H. T. Vo, S. P. Callahan, J. Freire, and C. T. Silva. Tackling the provenance challenge one layer at a time. *Concurrency and Computation: Practice and Experience*, 20(5):473–483, 2008.
22. Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, 2005.
23. A. Stuckenholz. Component evolution and versioning state of the art. *SIGSOFT Softw. Eng. Notes*, 30(1):7, 2005.
24. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
25. M. Treiber, L. Juszczyk, D. Schall, and S. Dustdar. Programming evolvable web services. In *PESOS '10: Proceedings of the 2nd International Workshop on Principles of Engineering Service-Oriented Systems*, pages 43–49, New York, NY, USA, 2010. ACM.
26. VisTrails. http://www.vistrails.org.