Using Provenance to Support Real-Time Collaborative Design of Workflows

Tommy Ellkvist[1], David Koop[2], Erik W. Anderson[2], Juliana Freire[1,2], and Cláudio Silva[2]

[1] Linköpings universitet, Linköping, Sweden
[2] University of Utah, Salt Lake City, UT, USA

Abstract. Because designing workflows is a notoriously difficult task, it often requires multiple users to collaborate. In such scenarios, sharing workflow evolution provenance in a timely manner is critical. We present an environment where collaborating users can see each other's changes in real-time. The synchronization of workflow evolution provenance is automatic, immediate, and unobtrusive, allowing users to see collaborators' changes as they are made. This enables a richer and fuller method of collaboration. We present the interface and algorithm for the synchronization and discuss common scenarios where this mechanism has been utilized.

1 Introduction

Scientific workflows are often used as a means to create computational processes that solve complex scientific problems in diverse areas. The design of workflows in multi-disciplinary research areas such as bioinformatics and environmental modeling often requires cooperation between multiple experts in different geographic locations. Currently, there are few tools available to support the collaborative design of workflows. Users are often limited to exchanging workflow specifications over e-mail. This process can be slow and tedious. In some cases, it may be possible to divide the work in such a way that collaborators can work independently and then combine their work for a final result. However, this assumes that a modular design is possible; in reality, workflows are often created by trial and error with many inter-dependencies.

To support the collaborative design of workflows, we propose a mechanism that allows collaborators to simultaneously work on a task and see each others's changes in real-time. With a group of users who are working on the same task, the changes made by each user are automatically propogated to the rest of the group. Note that we *do not automatically merge* changes like version control systems. Rather, we display each change as a new branch of exploration and allow the user to switch between branches regardless of who created them. Using workflow evolution provenance, for example the change-based representation for a collection of workflows [3], we can visually display a tree containing all contributions. This lets collaborators share and receive updates in real-time, while

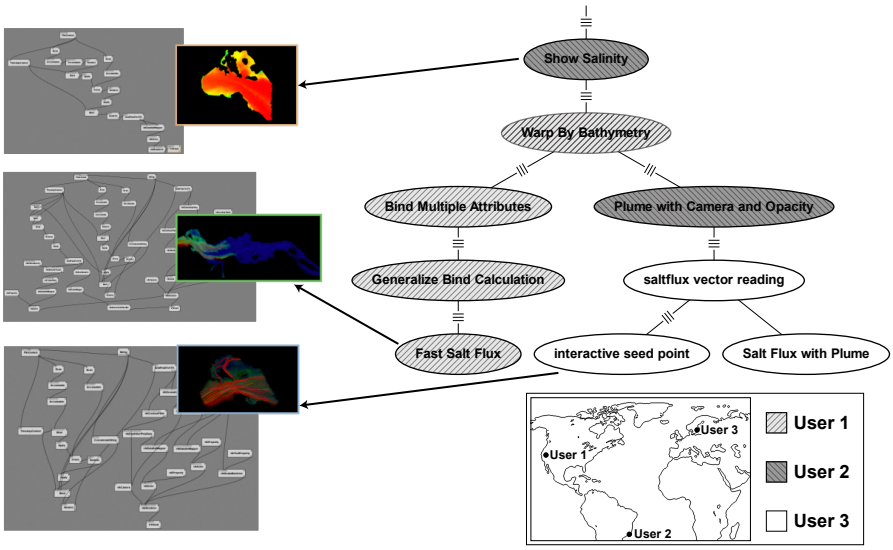

Fig. 1. A version tree containing a series of workflows that derive visualizations of the Columbia River Estuary. The visualizations have been created by collaborating users. Versions created by different users are represented using different colors.

at the same time giving them the option to selectively ignore updates they do not care about. In this paper, we describe an architecture that supports this functionality. We present a new algorithm for synchronization and discuss how it can be used in practice.

2 Architecture

In order to support real-time collaborative design workflows, we need a provenance architecture that supports a collection of versioned workflows and a centralized provenance repository that all collaborators can access. We require a versioning system because each user needs to know how their collaborators' work relates to their own. More importantly, we need to protect the users' work; we should not blindly erase or update their own changes. A centralized repository is needed to manage all the workflows and to provide the means for notifying collaborators when changes occur. The combination of these two methods not only allows users to efficiently share collections of workflows, but also enables them to see the entire history of the workflow specifications as they develop in real-time, regardless of how many users collaborate on the project.

Workflow Evolution Provenance. Because we expect to encounter a large number of changes to a workflow specification, especially in a collaborative environment, it can be inefficient to store specifications for all different versions of the workflows. The change-based provenance model [3] provides a concise representation for workflow evolution history. This model captures the changes applied to a series of workflows, akin to a database transaction log. As a user modifies

a workflow (e.g., by adding a module, changing a parameter or deleting a connection), the provenance mechanism transparently records each change action. We can then reconstruct any workflow specification by replaying the sequence of captured changes from an empty specification to the desired version.

The change-based model not only captures changes as a workflow evolves, but it also presents external changes to collaborators in a meaningful way. An important feature of this representation is that it can be visualized as a *version tree*, where each node corresponds to a workflow specification and each edge corresponds to the sequence of changes that transforms the parent specification into the child. Because the version tree captures *all* changes, users have great flexibility for exploring different alternatives without worrying about losing the ability to go back to a specific version. They can perform arbitrary undos and redos—any workflow version is easily recalled by selecting the corresponding node in the version tree. Additionally, users can easily see how their collaborators have taken different approaches to solving related problems and how their techniques relate to their own ideas. As discussed below, we leverage this layout to inform users of changes without forcing them to immediately consider or integrate those changes.

Centralized Repository. In order to efficiently capture and broadcast workflow changes, we use a relational database management system (RDBMS) for our centralized repository. We chose to use a RDBMS because these systems provide secure access protocols, support concurrent transactions from multiple users, and include trigger mechanisms for alerting users when the database is updated. These features are essential to ensure data consistency and to support real-times updates in our collaborative infrastructure. Other kinds of database systems that support these features could also be used in our infrastructure.

To use an RDBMS for our repository, we need to map the necessary provenance information to a relational schema. Because we use the change-based representation, a collection of related workflows is stored as a tree. This tree contains metadata and an ordered set of actions that correspond to user modifications to workflows. Each action, in turn, consists of a sequence of atomic operations. For example, a paste *action* that adds a set of modules and connections to an existing workflow contains a sequence of *operations*: `add module`, `add connection`, *etc.*. An operation, besides its data payload (e.g., module specification, connection specification, parameter value), includes metadata (e.g., the user who performed the action and annotations). Each of these entities (actions, operations, payloads) is stored in its own table, permitting a normalized (redundancy-free) representation. In addition to storing the changed-based representation of workflow evolution, the schema also supports explicit workflow specifications and workflow execution information. Execution information can be important when users are unfamiliar with the collection of workflows and wish to know which workflows are routinely used and which workflows were successfully executed.

3 Synchronized Design

One of the contributions of this paper is a new method for automatically capturing workflow changes performed by multiple users and alerting them about these changes immediately and unobtrusively. This allows users, in different geographically distributed locations, to collaboratively design and refine workflows, like in the scenario illustrated in Figure 1. We accomplish this by committing the local changes (performed by each individual user) to a centralized repository, sending the changes out from the repository to each collaborator, and adding the changes to each collaborator's local version tree. Note that we are not merging workflow specifications but synchronizing workflow evolution provenance. Each collaborator can continue their work and they need not even view the new changes. Before describing the implementation of our prototype, we describe the algorithm for synchronizing the version tree.

3.1 Algorithm

There are two key requirements for our algorithm. First, we need a way to save data from a local version tree to the centralized repository. Second, we need a way to load data from that repository to update the collaborators' local version trees. Below, we describe the mechanisms we developed to satisfy these requirements.

Recall that the version tree is induced by a set of actions A. Each action $a \in A$ has a unique identifier derived by the function $id : A \to \mathbb{N}$, where id assigns the smallest unassigned integer to a new action. This function is trivially monotonic: given $a_1, a_2 \in A$,

$$id(a_1) < id(a_2) \iff a_1 \text{ was added before } a_2$$

We will leverage this property to easily determine what has changed in a given version tree. Specifically, let

$$N(A) = \max_{a_i \in V} id(a_i)$$

be the largest action id in a set of actions A. Then, for two sets of actions, $A_1 \subseteq A_2$, the set of new actions, ΔA, is

$$\Delta A = \{a \in A_2 \mid N(A_1) < id(a) \leq N(A_2)\}$$

This means that we can efficiently determine which actions a user requires to update his version tree. If a user has copied all of the actions in the database up to id N_D, then we only need to copy actions a_i with $id(a_i) > N_D$ from the database. Conversely, if a user has already saved all actions up to N_L to the database, only actions a_i with $id(a_i) > N_L$ need to be sent to the database. Figure 2 shows a simple example of the steps of the algorithm.

Relabeling. Determining the set of new actions is easy when one of the two sets being compared is a superset of the other. However, when multiple users are collaborating, we might not be in this situation. Consider the scenario shown in Figure 3, where user A and user B made changes at the same time. Both clients will try to simultaneously save their actions to the database before being notified

Algorithm 1: Incremental Load Algorithm
Input: The local version tree V, id_V (the id function for V), the global-to-local id map M, and the centralized repository D.
Output: None. It updates both V and M in place.
LOAD(V, id_V, M, D)
(1) $max\_id \leftarrow$ Query V for the maximum id
(2) $A \leftarrow$ Query D for all actions with id $> max\_id$
(3) **foreach** a **in** A:
(4) Create a', a local copy of a
(5) $a'.id \leftarrow id_V(a)$
(6) $a'.prev\_id \leftarrow M_{local}(a.prev\_id)$
(7) Add pair $(a.id, a'.id)$ to M
(8) Add a' to V

of the other's changes. In each of their local version trees, they both have actions with id 7, but these actions are not the same. Assuming A's request gets to the repository first, her action will be given id 7 while B's action will become id 8. Thus, after pushing out the other's updates, A and B will have the same tree except that the ids of the nodes may differ.

Since an update of the ids in the local version tree might interfere with a user's current work, we choose to maintain a set of local ids that can be mapped to the global repository ids. Specifically, we maintain a bijective map

$$M : id_{global} \leftrightarrow id_{local}$$

Let M_{local} denote the reverse mapping from global to local and M_{global} denote the forward mapping from local to global. All user operations will be accomplished using the local ids, but whenever we need to save to the centralized repository, we translate everything to the global set of ids. Figure 3 shows an example of this relabeling.

Beyond Actions. As described earlier, an action contains metadata and a set of atomic operations. The metadata and the atomic operations, in turn, contain their own ids and may also include references to other entities. Thus, the relabeling of an action needs to update these references as well. For example, each action stores both its own id (*action.id*) and its parent id (*action.prev_id*). If we update the id of the action referenced by *action.prev_id*, we also need to update the *prev_id* field. The same is true for child objects. Suppose the connection in an `add connection` operation references the two modules it connects by id. If we remap the id of one or both of those modules in an `add module` operation, we need to update the ids in the `add connection` operation as well. This requires an ordering that respects the properties being updated; we impose an explicit order on modules and connections so that all modules are relabeled before connections to ensure all references are updated.

Algorithm Specifics. We combine the method for determining new actions with our relabeling strategy to obtain robust algorithms for incrementally load-

Algorithm 2: Incremental Save Algorithm
Input: The local version tree V, id_D (the id function for D), the global-to-local id map M, and the centralized repository D.
Output: None. It updates both V and M in place.
STORE(V, id_D, M, D)
(1) $max\_id \leftarrow$ Query D for the maximum id
(2) $A \leftarrow$ Query V for all actions with id $> max\_id$
(3) **foreach** a **in** A:
(4) Create a', a global copy of a
(5) $a'.id \leftarrow id_D(a)$
(6) $a'.prev\_id \leftarrow M_{global}(a.prev\_id)$
(7) Add pair $(a'.id, a.id)$ to M
(8) Add a' to D

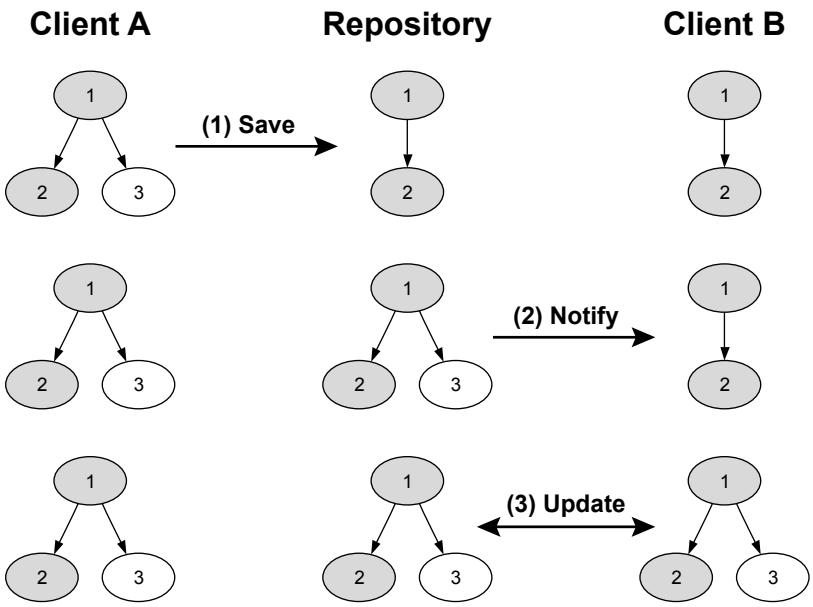

Fig. 2. The synchronization algorithm. Client A creates a new change (labeled as version 3). This new version is automatically saved to the repository (Step 1). Whenever the repository is updated, it notifies all clients of the new change (Step 2). All clients (including Client B) then incremenetally update themselves (Step 3).

ing from and saving to a database. Algorithm 1 describes the loading algorithm and Algorithm 2 summarizes the saving algorithm. In each algorithm, we use either the database or local version tree to update the other depending on the direction, ensuring that new ids are assigned, existing ids are remapped, and the global-to-local mapping M is updated. Note that all entities are updated in place, copying only the (new) required information from one side to the other.

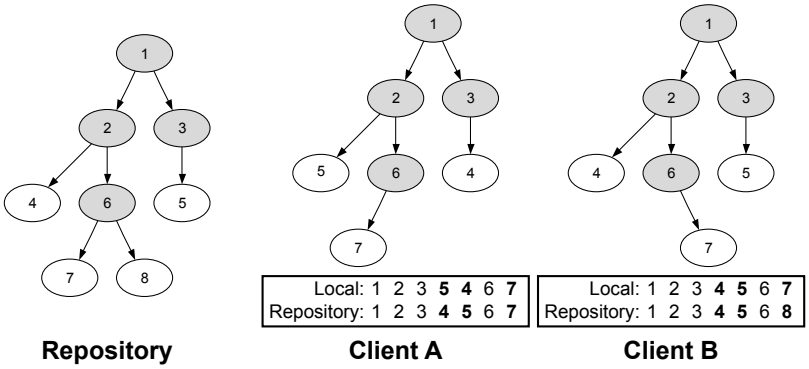

Fig. 3. Relabeling. Because two users may make updates at the same time or may temporarily lose their connections with the repository, the ids of their nodes may not correspond with the repository's ids. To solve this problem, each client stores the tree according to its own local ids and maintains a map to the repository's global ids.

3.2 Implementation

We have implemented the synchronization mechanism on top of the VisTrails system (http://www.vistrails.org). The implementation consists of a client/server architecture shown in Figure 2. The server-side is a MySQL database that stores version trees. Users can create synchronization sessions through the user interface (see below). The standard VisTrails database schema has been extended to store information about synchronized sessions. This information includes the ids of synchronized version trees, user ids, IP addresses, and port numbers. A database trigger uses this information to notify clients when relevant updates are available. The notification is done by an external MySQL function that uses a socket to connect to the client. The message to the client includes the version tree id number so that the client can request the updates for that version tree. Note that messages about changes to a given version tree are sent to all users using that version tree, except to the user whose changes activated the trigger.

The client-side application is a modified version of VisTrails; the modifications include code for performing incremental updates and saves against the database and for receiving notification messages from the database. Because the system contains a controller object for each version tree, we use it to monitor these notifications and start update procedures. Because the controller is linked to the GUI, we also need to redraw the version tree whenever synchronization modifies the tree.

To setup synchronization, users need to select (or create a database) to serve as a centralized repository. This database must have the schema as outlined above and the synchronization triggers that send the update notifications. Once the database is in place, users connect to the database and select the version trees they want to share. After that, the synchronization (sync) mode can be enabled with the push of a button. From that point on, the version tree will be

kept in sync with the central repository and the other users. To help distinguish between versions, those created by other users are shown in blue while a user's own versions are highlighted in orange.

3.3 Issues

Mutable Objects. The monotonicity of the version tree is required for the synchronization process. Change actions and operations are immutable: they are never modified after they are stored in the repository. Thus, the system only needs to check for *new* objects in order to perform synchronization. There are, however, *mutable* objects associated with actions for which this optimization cannot be applied. For example, VisTrails has *version tags* and *version annotations* associated with workflows that can be modified, and these modifications are not saved as actions. Version tags assign text labels to workflow versions while version annotations store general notes about the version. Because changes to these objects are non-monotonic (and destructive), *all* objects must be saved and loaded during each incremental load/save. Locally, we can keep a flag that indicates whether or not the entity changed so that we only need to save it when it does, but the same cannot be done for the global repository. Nonetheless, since the volume of mutable data is small, we copy all instances during an incremental load.

Integrating Changes. One nice feature of our synchronization framework is that it does not require the user to integrate another user's changes. However, consider the situation where two users (A and B) are working on a similar problem, and they have attacked different pieces of it from a common starting point. Each has seen that the other has made changes, but they wanted to finish their own piece. Later, when they decide to integrate these changes, user A can switch to B's version and make the changes applied in her own version. A more efficient alternative would be for user A to use the *analogies* mechanism [10] implemented in VisTrails to automatically apply the changes from one branch to another.

Local parameters. Workflows may not always have the same meaning to all users, and they may disagree about certain parameter settings or methods used. For example, an input filename parameter may differ between two users because the users store the file in different disk locations. Currently, the only way to deal with such local parameter settings is to create a different version for each set of parameters. This means that a change in one user workflow will not propagate to the other version, which is not desirable. A solution to this problem could be to separate the shared workflow from the local settings creating a division of the workflow in some way.

Data sharing. The ability to share data is an important part of collaboration. For workflows, you may want to share output data as well as input and intermediate results. This can be done with a data pool which maintains up-to-date data items created by the users. This would make it possible for users not only to

see each other's results, but also use the data as inputs to other workflows. The COVISA project[12] implements this kind of data sharing. Users can exchange data and directly use them in their pipelines. Another system that implements the idea of a data pool is the *Data Playground*[4]. The Data Playground provides a workflow editor that is highly data centric, letting users view and import data while they compose workflows that in turn create new data items. This gives the users control over their data while they experiment with different data manipulation operations. The prototype only works for one user but it shows how a data centric view can be used in collaborative workflow design.

Module packages. A requirement for users to be able to share workflow specifications is that they both use the same repository of module packages. Module packages contain sets of modules that perform similar functions, much like web services. If one collaborator is missing a module, a workflow containing that module can not be executed. For collaborations that require many different packages and libraries, an effective mechanism is needed for sharing. For example, through the use of public repositories or automatic methods for users to import module packages from other users as they are required. The packages are often platform specific and versioned, so finding the right package is not trivial. This requires packages to use a good version scheme, with possibly backward-compatible packages. There also needs to exist different versions for different platforms so that the users platform can be identified and the correct package used. Another way to handle module sharing is to use shared computing infrastructure, such as the TeraGrid (http://www.teragrid.org), which can provide a comprehensive set of packages.

3.4 Discussion

While there are many systems that provide mechanisms to deal with the difficulties associated with the collaborative modification of files, they are not built to handle structured information like workflows. For this reason, many workflow systems lack comprehensive version control for their workflow specifications.

Many systems have been developed with the singular purpose of providing version control. Software such as SVN [8], CVS [1], and Visual Source Safe [7] are optimized to robustly handle the version control requirements associated with source code. Unfortunately, when dealing with workflow descriptions, the standard merge operations common to text files are inadequate and require specialized processing. A second issue is that these systems require users to *manually* perform check-ins and check-outs in order to synchronize versions. Finally, users are often required to merge their changes with older changes, making it more difficult to explore new directions.

We address the shortfalls of standard version control with our method based on synchronizing workflow evolution provenance. Using this approach, workflow descriptions can be analyzed and modified to provide a truly multi-user, collaborative environment, in real time. These modifications provide the basis for version control of rapidly evolving, collaboratively created workflows. The intu-

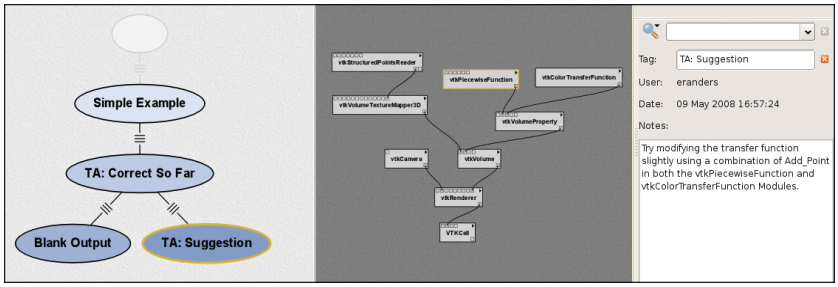

Fig. 4. An example of a TA session. The TA can highlight interesting versions in the students version tree as well as create new versions that explain some part of the workflow design.

itive system allows closer collaboration between users by immediately alerting all users of each other's changes.

4 Use Cases

Collaboration between two or more parties plays an important role in scientific discovery and in education. By carefully examining the working process of existing collaborative research projects, we have been able to design a system that not only respects individual working habits, but also strengthens and enhances the interaction among multiple users engaged in collaborative efforts. Here, we explore the benefits of real-time, synchronous collaborative workflow design.

Collaborative Design as a Teaching Aid. Many institutions of higher education offer a wide range of courses that utilize workflow systems. For example, in Scientific Visualization courses, the Visualization Toolkit [6] (VTK) is widely used to teach different visualization techniques to the students. Instructors use VTK to introduce various topics to the students *by example*, while the students use the library to explore the advantages and caveats associated with the various techniques they learn.

A first experience in using VisTrails to encapsulate VTK pipelines used in a Scientific Visualization course was very successful and showed that the reproducibility and sharing enabled by provenance is very beneficial in a teaching environment. However, even when using a provenance-aware system, a large amount of work was necessary to assist students with the various assignments. In these cases, the Teaching Assistant (TA) had to meet individually with each student to help solve the problems they had.

By providing TA's with a system capable of synchronous, collaborative design of workflows, the time necessary to assist students can be greatly reduced. Instead of the students relying on restrictive office hours to get face-to-face help, they are able to get assistance from the TA as they work from their workstation (see Figure 4). This decreases the amount of time the students need to

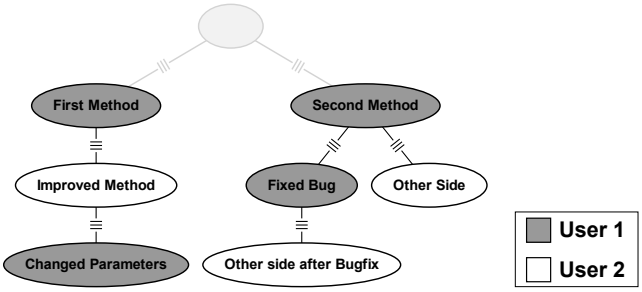

Fig. 5. An example of collaborative design. here, two persons have built on each others workflow specifications, leading to incrementally better results.

spend waiting for help and allows the TA to interactively explain the reason the student's workflow was incorrect. Coupled with an instant messaging (IM) program, this collaborative session greatly increases the number of people the TA is capable of helping in a given amount of time.

Collaborative Design in Multi-disciplinary Research. In today's scientific community, it is rarely the case that novel scientific discoveries can be made by a single person. Unfortunately, in many instances of close collaboration, the various domain experts are unable to work in the same location. These types of relationships benefit greatly from the ability to concurrently modify a given workflow description.

An example of the advantages gained from collaboratively designed workflows can be seen in collaborations between the authors at the University of Utah and researchers at the Center for Coastal Margin Observation and Prediction (CMOP).[3] CMOP scientists, located in Oregon and Washington, often spend a significant amount of time describing the various processing and analysis methods they employ to understand their data. While in many cases e-mail is satisfactory for sharing knowledge with collaborators, in some situations, a more immersive collaborative workspace is required.

When a task relating to a specific researcher's area of expertise is being considered, it is often necessary to synchronize processing workflows to arrive at a desired result. By allowing scientists at the CMOP centers in Oregon to work synchronously with researchers at the University of Utah, the critical task of communication is enriched. Instead of relying on e-mail and telephone conversations to ask important, and often time-consuming, questions, scientists can explore *and fix* each others processing and parameterization errors in real-time. This degree of collaborative design reduces the number and severity of communication-based misunderstandings as well as increases the level of productivity of everyone involved in the project.

[3] http://www.stccmop.org

5 Related work

This paper presents, to the best of our knowledge, the first proposal for an infrastructure that supports real-time collaborative workflow design.

There are existing mechanisms that can be used for collaborative design of workflows. One of the most general and common methods of real-time collaboration is through remote desktops like VNC [9]. By using this in the design of a workflow, users can see each others operations like dragging modules around and creating connections. But for more efficient modes of interaction, both users need to be in control simultaneously, and be able to choose whether to take notice of other users activities. In addition, provenance information would be lost, since it would not be possible to distinguish changes performed by different users.

A related area is that of collaborative visualization such as the COVISA project[12] and NoCoV [11]. COVISA enables several modes of collaboration like sharing data, sharing control of parameters and instructor driven collaboration where one user is in control of another user's pipeline. NoCoV enables users to collaboratively edit a pipeline consisting of instances of *Notification Web Services*. Both of these systems enables collaboration in the creation of the visualization pipeline but they do not support the exchange or existence of different versions of the pipeline.

The use of real-time collaboration has been explored in other areas. *Co-browsing* [2] enables multiple people to browse the by sharing a Web browser view and following links together. Similar to VNC, co-browsing is useful when a user wants to guide another through a browsing session. However, unlike VNC where the whole desktop is shared, in co-browsing users only share a browser view. Co-browsing can thus be more efficient, since only clicks withing a browser view need to be propagated to the users.

A more indirect way of sharing workflows is through public repositories, like myExperiment [5] and Yahoo! Pipes [13], that have become available recently. These repositories foster the re-use of knowledge. They provide search interfaces that allow the users to locate workflows that solve a particular task, and then integrate these workflows into their own. The synchronization infrastructure we propose could potentially be a useful feature offered by these sites.

6 Conclusion

In this paper, we described an infrastructure that supports real-time collaborative design of workflows. This infrastructure can be integrated with any workflow system that captures workflow evolution provenance. Our implementation of the synchronization mechanism on top of the VisTrails system shows that workflow systems can be a powerful tool for real-time collaboration. Users can collaborate efficiently and effectively, exploring different branches and taking advantage of each other's progress. Together with techniques for data sharing and remote execution, this enables efficient creation of complex workflows.

By leveraging the concise representation of workflows provided by the change-based provenance model, synchronization is efficient: only incremental changes need to be propagated to collaborating users. However, further experiments are needed to assess the scalability of the current implementation.

We believe that our provenance-based synchronization mechanism can be applied to applications other than workflows. Combined with techniques to visualize provenance information, this mechanism can serve as a powerful platform for collaborative design in general. Users can share their work effectively while inspecting each other's contributions. The application of our synchronization infrastructure in other areas of computational design is a direction we plan to pursue in future work.

Acknowledgements

Our research has been funded by the Department of Energy SciDAC (VACET and SDM centers), the National Science Foundation (grants IIS-0746500, CNS-0751152, IIS-0713637, OCE-0424602, IIS-0534628, CNS-0514485, IIS-0513692, CNS-0524096, CCF-0401498, OISE-0405402, CCF-0528201, CNS-0551724), and IBM Faculty Awards (2005, 2006, 2007, and 2008).

References

1. P. Cederqvist et al. *Version Management with CVS (for CVS 1.11.6)*, 1993.
2. A. Esenther. Instant co-browsing: Lightweight real-time collaborative web browsing, 2002.
3. J. Freire, C. T. Silva, S. P. Callahan, E. Santos, C. E. Scheidegger, and H. T. Vo. Managing rapidly-evolving scientific workflows. In *International Provenance and Annotation Workshop (IPAW)*, LNCS 4145, pages 10–18, 2006.
4. A. Gibson, M. Gamble, K. Wolstencroft, T. Oinn, and C. Goble. The data playground: An intuitive workflow specification environment. In *E-SCIENCE '07: Proceedings of the Third IEEE International Conference on e-Science and Grid Computing*, pages 59–68, Washington, DC, USA, 2007. IEEE Computer Society.
5. C. A. Goble and D. C. D. Roure. myexperiment: social networking for workflow-using e-scientists. In *WORKS '07: Proceedings of the 2nd workshop on Workflows in support of large-scale science*, pages 1–2, New York, NY, USA, 2007. ACM.
6. Kitware. The visualization toolkit (VTK). http://www.kitware.com.
7. Microsoft Corporation. Managing projects with Visual SourceSafe. Redmond, Washington, 1997.
8. M. C. Pilato, B. Collins-Sussman, and B. W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, Inc., June 2004.
9. T. Richardson, Q. Stafford-Fraser, K. R. Wood, and A. Hopper. Virtual network computing. *IEEE Internet Computing*, 2(1):33–38, 1998.
10. C. E. Scheidegger, H. T. Vo, D. Koop, J. Freire, and C. T. Silva. Querying and creating visualizations by analogy. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1560–1567, 2007.
11. H. Wang, K. Brodlie, J. Handley, and J. Wood. Service-oriented approach to collaborative visualization. In *Proceedings of UK e-Science All Hands Meeting 2006*, pages 241–248. National e-Science Centre, 2006.

12. J. Wood, H. Wright, and K. Brodlie. Collaborative visualization. In *VIS '97: Proceedings of the 8th conference on Visualization '97*, pages 253–ff., Los Alamitos, CA, USA, 1997. IEEE Computer Society Press.
13. Yahoo! Pipes. http://pipes.yahoo.com [10 March 2008].