OntoDesk: Ontology-Based Persistent System-Wide Undo on the Desktop

David Nemeskey, Buntarou Shizuki, and Jiro Tanaka

Department of Computer Science, University of Tsukuba
nemeskey@iplab.cs.tsukuba.ac.jp,
{shizuki,jiro}@cs.tsukuba.ac.jp

Abstract. Recovery is an important aspect of user experience. However, current desktop environments lack a system-wide undo facility. OntoDesk is an ontology-based experimental desktop system that offers this feature. Ontology is used to model the semantic relationships between parts of the system. OntoDesk assembles a global action history of application use. With this information, it provides undo/redo for any part of the system, including applications without native recovery. The framework allows developers to add advanced features to their applications, and it allows users to explore the system with confidence, knowing that their actions will be reversible.

Keywords: OntoDesk, ontology, OWL, system-wide undo, persistent undo, application, action, global history, session management.

1 Introduction

User interaction history plays an important role in interactive systems. Most desktop applications allow the user to undo past actions, enabling the user to recover from errors and to explore application functions without hesitation [1].

However, an action history usually belongs to a particular application: other programs cannot access it, and it is lost when the application is closed. Furthermore, even modern operating systems do not support undoing system-level events, such as starting and stopping an application and file creation.

These limitations seriously hinder the utility of recovery in an environment where a single mouse click is enough to make a mistake. For instance, the user can accidentally change the desktop background while trying to save a picture in a browser, since the options are close in the context menu. It is also not uncommon that the user closes an application accidentally. While certain applications can reestablish their previous state on restart, others forget the navigation history. Consequently, the lack of persistent undo history means that users cannot undo errors from a previous session; the prior state must be reestablished manually.

Joyce, a distributed system framework provides persistent, system-wide undo [2]. The undo mechanism is based on manually-defined dependencies between the actions of applications. While this method works for standalone applications, tasks that involve several applications requires that inter-application dependencies are defined. This is not feasible on the desktop, where applications come from many sources.

J.A. Jacko (Ed.): Human-Computer Interaction, Part I, HCII 2009, LNCS 5610, pp. 890–899, 2009.

In this paper, we propose an ontology-based framework that provides persistent system-wide undo, eliminating all the aforementioned limitations of recovery. The system objects, such as files and documents, as well as applications and user actions are described by an ontology. This allows us to build a model that decouples actions from their applications on an abstract level. Our framework tracks the actions and maintains a persistent, global action history.

Using an ontology delivers several benefits. Firstly, system objects and actions are defined in an abstract, unified manner, regardless of implementation details. This allows the undo model to be general and applicable to any system object: users can undo anything from modifications to a file, to closing of an application or even changing of the desktop background. Secondly, as the ontology is a text-based data model, it frees the framework from programming languages and toolkits. As a result, existing applications can be integrated into the framework. Finally, we can infer inter-application dependencies even for applications that were not designed to work together.

2 Approach

Conventionally, the actions that the user can invoke in an application are not visible to the system. As a result, the system is ignorant of user actions in any specific application. Conversely, events and logs reported by applications to the operating system are usually insufficient for recovery and too low-level to present to the user.

We intend to rectify this situation by making actions first-class citizens within the framework. Applications define the available high-level actions. These are the *semantic* actions that users consider when using the computer, such as "insert text to a document," "send a mail," or "close a program." Applications report the actions they execute to our framework, which builds a persistent, global history list of these action records.

The heart of our framework is the ontology, "an explicit specification of the concepts in a domain and the relations among them" [3]. Here, the ontology serves as an intelligent registry where applications and other parts of the system, such as the runtime environment, publish their capabilities. It also tracks all system *resources*, such as files, documents and running application processes; they are the objects that live in the system and serve as parameters for actions. Resources also have *states*, which represent the data (or literally, the state, in case of application processes) of the resource at a given time. The history is stored in the ontology as well. Applications include an ontology file that describes their capabilities. This file is merged to the system ontology when the application is installed.

Our research focuses on the controlling aspects of desktop systems. The framework models the control flow of a typical desktop environment. Data and content are modeled only to the extent that is necessary for the undo facility. File metadata, interrelations between documents, etc. are not supported. Data semantics are already being investigated by other projects, such as Nepomuk [4].

2.1 Architecture of the Framework

The main component of the framework is the *Ontology Server*, which runs as a system service. All ontology queries and modifications are carried out by the server.

Other processes can access it only through the interface provided by the server. Applications communicate with the server via IPC.

When the user executes an action in an application, the description of the action is sent to the server, which stores it in the ontology. Undo and redo are reported in the same way. The server can also instruct an application to execute one of its actions, or perform an undo or redo operation (see below).

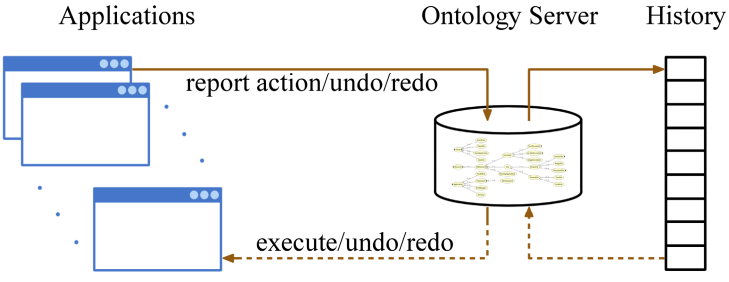

Fig. 1. Control flow of the framework

2.2 Undo Support

A system-wide undo facility faces challenges that do not appear in traditional applications. Firstly, the global history may contain non-undoable actions, such as sending a mail. Secondly, finding the specific action to undo in a global history would be overwhelming, requiring system-generated *resource-local histories* including only those actions that affected a particular resource. This is complicated by actions which can affect multiple resources. Lastly, effects of the undo operation should be localized: parts of the system unrelated to the undone action should not be affected. These features demand a more sophisticated undo model.

Undo Model. Most of today's applications provide *linear undo*: actions are stored in a single history list, and undone in reverse temporal order. In our case, however, actions related to a certain resource are scattered in the global action list. Similar challenges are encountered in multi-user undo, and necessitate a *non-linear undo* model, where actions can be undone regardless of their placement in the history [1].

Another contrast to the traditional undo model is that actions may depend on the result of other actions. For example, actions that belong to an application can only be executed if the application has been started before. Two strategies exist for the case when an action, whose results are used by other actions, is selected for undo. *Direct selective undo* [1] forbids such operations. *Cascading selective undo* [5] undoes dependent user actions until a meaningful state is reached. Since the former approach would be very limiting, we have opted for cascading selective undo in our framework.

When an action is selected for undo, the system first determines its *undo closure.* It includes all actions that directly or indirectly depend on resources created or changed during the execution of the selected action. Then all actions in the closure are undone in reverse temporal order. If any action in the closure is not undoable, the whole undo request is rejected. This process ensures that (1) the system will be in a consistent state after the undo operation, and (2) unrelated actions are not affected.

Although a form of selective undo is employed "under the hood," we decided to allow only linear undo on the UI. Users are not familiar with the selective undo mechanism; neither are there applications that support it natively. Further, linear undo fulfills the *stable execution property*: the command is always undone in the state that was reached after execution [1]. This property greatly simplifies reasoning in the ontology, as there is no need to verify if an action is undoable in the current state.

Application- and Server-based Undo. Our framework provides three methods of undoing actions. Even if an application does not implement undo (at all, or for a set of actions), the Ontology Server may execute the undo based on the information present in the ontology. Hence, we differentiate between *native* (application-based) and *server-based* undo.

In the case of native undo, the undo facility is implemented in the application itself. When the user issues an undo or redo command, the application executes it and reports it to the server. The server then modifies the history accordingly.

Server-based undo is implemented using two strategies: *inverse actions* and *partial checkpoints* [6]. These methods are chosen only if native undo is not available.

Applications can define inverses for their actions in the ontology. When an action with an inverse is selected for undo, the server assembles a message for the inverse action, and fills its inputs from the data of the original action. It then sends the message to the application and instructs it to execute the inverse action.

Our framework also allows actions to be defined as *checkpoints* for resources. This tells the system that the resource is in an easily reproducible state after the action is executed. For example, a *Save File* action serves as a checkpoint for documents; *Start Application* action for an application.

Using checkpoints, the server can undo even non-native, non-invertible actions. First, the system looks for the latest checkpoints of all input resources of the selected action. It then re-executes all commands from the checkpoints up to, but not including, the action, thereby resetting the state of all related resources to their condition prior to the action.

3 Ontology Design

Our ontology language of choice is OWL-DL [7]. It is based on description logic, thus not dissimilar to an object oriented language [8]. Objects in the domain are represented as *individuals*; these are then arranged into a *class* hierarchy. Individuals have *properties*, which can refer to other individuals or simple data types. To represent actions, we use OWL-S, a vocabulary for Semantic Web Services [9]. Actions in our ontology are OWL-S *Service*s.

Two ontology files are used by the Ontology Server. The *definition ontology* contains the type hierarchy and the individuals that represent system-level entities, including the registered actions and applications. The *history ontology* contains the history list and all related data.

3.1 Main Classes

The ontology defines three main categories: *Resources, Actions* and *Applications*. Actions are *provided by* the Applications. For example, text editors provide the *Insert*

Text action, which inserts text into a document. A special application type, *Framework* provides the system-level actions.

The currently supported resource types are *Document*, *File* and *RunningApplication*. A RunningApplication object represents a running process. Documents represent meaningful data, such as a body of text or an image. They are *stored in* Files. A file may contain several Documents, such as a multimedia file that contains video and audio streams. Data that has not yet been saved into a file, e.g. when the user starts drawing a picture in an image editor, can also be represented. In such a case, the Document is *stored in* the RunningApplication instance. Modifications to the data only affect the Documents, and the changes are visible in the File only after a *Save File* action is executed (see Fig. 2).

3.2 Undo Handling

The undo options discussed in Section 2.2.2 are represented as follows. The *supportsUndo* property shows if an application supports undo. Actions can have the following properties:

- The *undoable* property shows if the action is undoable;
- The *nativeUndo* property shows if the actions is undoable natively;
- The *inverseAction* property is used to define the inverse of the action;
- The *checkPoint* property declares that the action is a checkpoint for the related resource.

If the *nativeUndo* property is true, native undo is performed on request. If false, but the action has an inverse (the action the *inverseAction* property refers to), the inverse action method is used. Otherwise, if, for every resource-type input parameter of the action, there is at least one action in the history whose *checkPoint* property refers to the input resource, the checkpoint method is selected. If none of these conditions are fulfilled, or the *undoable* property is false, no undo is possible for the action.

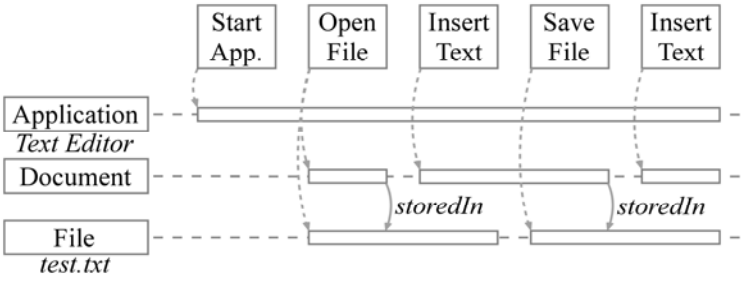

Fig. 2. Shows how the resources (left) and their states (narrow bars) change when the user executes an action sequence (top). The dotted arrows indicate which actions result in new states for the resources. The meaning of the storedIn property is as described above. The storedIn property arrows between the application and document states are omitted for brevity.

3.3 Action History

Our handling of action history is similar to the model described in [10]. The history consists of *ExecutedAction* objects. They contain a reference to their action type, and store the actual values for the input and output parameters of the action. These values can be Resources in the ontology or raw data (strings, numbers, etc.).

Resources are added to the ontology as parameters of the ExecutedAction objects. The only resources in the history ontology are those that are referenced by the actions; i.e. not all files in the system are represented in the ontology. In this way, all information necessary for undo is stored in the ontology, yet the number of objects – and therefore overhead – is minimized.

The ontology must reflect a constantly changing system state. The framework employs a very simple time scale, where the system state changes only as the result of actions. Because all actions are saved as ExecutedAction objects, they can serve as time points. Changes to Resources are modeled with *State* objects. Every Resource is assigned a State when it is created. Actions that modify the data associated with a Resource (as opposed to reading it; e.g. the *Save File* action modifies the state of the file, but *Load File* does not) report this fact by requesting a new State for it. Relations between resources are represented by properties between their States. This enables the ontology to represent relations whose validity changes with time, such as "after the *Save File* action, the document stored in the file is the same as the one loaded into the application (see Fig. 2)."

4 Implementation

To test the feasibility of our approach, we implemented a mock desktop, OntoDesk. Our experiences with the system allowed us to examine the requirements for integrating applications to our framework, and the effort required.

4.1 OntoDesk

OntoDesk is a desktop simulator written in Java. It supports typical desktop features like wallpapers, a start menu and window management. Applications are displayed in internal frames. Currently, OntoDesk includes the following applications:

- A file manager,
- *Image Editor* – an image viewer/editor for the user to draw simple shapes,
- Two text editors, *Text Editor* and *Simple Editor*. Although they have the same features, *Simple Editor* relies entirely on the framework to provide undo.

The Ontology Server is implemented as a system thread and communicates with applications by socket interface. The system ontology is accessed via OWL API [11].

The undo operation should be easily detected and specified, and the state resulting from the execution thereof should be easily predicted [12]. Though our system allows only linear undo, all resources have their own histories, requiring an easy-to-use interface to the system-wide history facility. OntoDesk introduces the *History Viewer* tool, which allows the user to choose a resource or a running application and display its

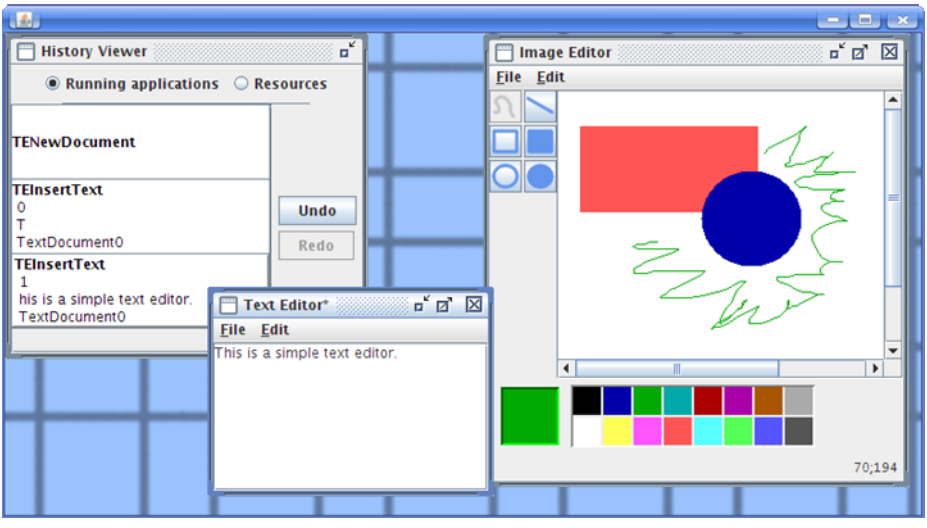

Fig. 3. OntoDesk with Image Editor and Text Editor open (right). The History Viewer (left) is displaying the actions executed in the Text Editor.

action history. Undone operations are displayed in a lighter color. The user can also undo and redo selected actions. If the selected action is not last in the history, all subsequent actions are also undone.

All applications use our framework to access the history of their resources and to implement undo/redo. The file manager can display the undo history of a file in the *History Viewer*. When a file is opened in a text or image editor, previous modifications are loaded with the file, and they become available to the user. Furthermore, the user can undo the closing of applications, in which case OntoDesk restores the application and its documents to the state it was in before it was closed.

4.2 Requirements for Integration

In this chapter, we review the requirements of integration and the effort needed.

Ontology. Applications that want to utilize the framework's capabilities must define an ontology including the:

- Actions they provide. Actions defined by other applications or the system itself may be reused. In OntoDesk, the *Simple Editor* uses the same actions as the *Text Editor*; it is only the implementation that is different.
- Resource types that are specific to the application, such as custom file types.

Changes in the Application. Integration to the framework also requires that the code base of the application be extended with the following three components:

- A *communication interface* to the Ontology Server. As this component is completely generic, it can be abstracted into a library.

- A *command interface* that acts as a mediator between the Ontology Server and the application. It reports executed actions to the Server and forwards its requests to the appropriate parts of the application.
- *Callback hooks* need to be added to the already existing source code to notify the command interface about user actions and their parameters. These hooks are typically placed in the event handlers of the application's interfaces (UI or otherwise), and are generally short (a few lines per event).

A Case Study Using OntoDesk. All applications in OntoDesk had been developed as regular applications before they were adapted to our framework. This proves the feasibility of integration. By comparing the application source code before and after the adaptations, we can measure the effort required for integration.

OntoDesk provides the communication interface as a Java library. The command interface is provided as a system class, which implements basic event reporting. Client applications extend this class through subclassing to fit to their needs. Table 1 summarizes the additional modifications.

As we can see, while the framework requires some boilerplate code, only a moderate number of additional lines are required for new actions. Although the percentage of framework-related code may seem high, it must be noted that the applications are very simple. The complexity of real-life applications dwarfs the 16-20 lines per action required. Moreover, utilizing the framework can even help reducing code size: by delegating undo to the framework, Simple Editor's code base became 23% shorter than that of Text Editor (*).

Table 1. Source code modifications required by the integration in OntoDesk. LOC = Lines of code, excluding comments. ΔLOC shows the number of extra lines required for the command interface and callback hooks. Avg. LOC/action shows the average LOC needed per action.

Application	# of actions	LOC	ΔLOC	Avg. LOC/action	Framework code %
Text Editor	4	834	181	20	17.8%
Simple Editor	4	634*	159	18	20%
Image Editor	3	858	148	16	14.7%
File Manager	1	996	76	16	7%

5 Advantages of Our Framework

A persistent, system-wide undo benefits the user by removing today's recovery limitations. This framework also offers advantages to developers.

In OntoDesk, programmers can utilize the built-in History Viewer tool. Since it provides an application-independent way of displaying action histories, developers can choose to omit the undo/redo options from the user interface of their applications, and rely on the History Viewer instead. OntoDesk's *Simple Editor* uses this approach.

Server-based undo can be used to add undo capabilities to an application merely by defining the inverse and checkpoint properties for its actions. In OntoDesk, several applications use this approach. *Image Editor* and *Text Editor* natively support undo for document editing. However, it is also possible to undo the New Document or Load File actions, a feature generally lacking in applications. Here it is achieved by

server-based undo. Furthermore, *Simple Editor* does not support native undo at all, yet, with OntoDesk, it has feature parity with *Text Editor*.

In addition, the persistent undo mechanism can be used to implement session management. OntoDesk handles system sessions in this way: when the user closes the main window, all applications are closed. When OntoDesk is restarted, these actions are undone, and the system returns to the state it had before the user left it.

6 Future Work and Conclusions

In this paper we have proposed a framework that allows persistent, system-wide undo on the desktop. Allowing applications to describe the actions they provide in an ontology enabled our system to create an abstract model of the capabilities of its components. This information can be used to provide a global action history. Our framework can be applied to existing applications with minimal modifications. OntoDesk, an experimental desktop, showcases the potential of our approach.

A global action history has uses besides beyond undo/redo. It could serve as a basis of a system-wide macro facility [13]. By enriching the semantic information in the ontology, a Programming By Example system could also be based on our framework. We would like to explore these possibilities in future versions of OntoDesk.

Our final goal is to test the feasibility of our framework in a real-life environment by adapting our framework to an existing application or desktop environment.

References

1. Berlage, T.: A selective undo mechanism for graphical user interfaces based on command objects. ACM Transactions on Computer-Human Interaction 1, 269–294 (1994)
2. O'Brien, J., Shapiro, M.: Undo for anyone, anywhere, anytime. In: Proceedings of the 11th workshop on ACM SIGOPS European workshop, vol. 31 (2004)
3. Noy, N.F., et al.: Creating Semantic Web Contents with Protégé-2000. IEEE Intelligent Systems 16, 60–71 (2001)
4. NEPOMUK – The Social Semantic Desktop, `http://nepomuk.semanticdesktop.org/`
5. Cass, A.G., Fern, C.S.T.: Modeling dependencies for cascading selective undo. In: IFIP INTERACT 2005 Workshop on Integrating Software Engineering and Usability Engineering (2005)
6. James, E., Archer, J., Conway, R.W., Schneider, F.B.: User Recovery and Reversal in Interactive Systems (1981)
7. OWL Web Ontology Language Reference, `http://www.w3.org/TR/owl-ref/`
8. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The description logic handbook: theory, implementation, and applications. University Press, Cambridge (2003)
9. OWL-S: Semantic Markup for Web Services, `http://www.w3.org/Submission/OWL-S/`
10. Zhou, C., Imamiya, A.: Object-based nonlinear undo model. In: Proceedings of the 21st International Computer Software and Applications Conference, pp. 50–55 (1997)

11. The OWL API, http://owlapi.sourceforge.net/
12. Masuda, H., Imamiya, A.: Design of a graphical history browser with Undo facility, and visual search analysis. Syst. Comput. Japan 35, 32–45 (2004)
13. Myers, B.A., Kosbie, D.S.: Reusable hierarchical command objects. In: Proceedings of the SIGCHI conference on Human factors in computing systems: common ground, pp. 260–267 (1996)