Undo for anyone, anywhere, anytime

James O'Brien, Marc Shapiro
Microsoft Research
7 J J Thompson Ave.
Cambridge, UK
i-jameso@microsoft.com

Abstract

Computer systems are complex and unforgiving. Users need environments more tolerant of errors, allowing them to correct mistakes and explore alternatives. This is the aim of Joyce. Joyce records application usage across the system in such a way that the semantic relationships between individual operations are preserved. Using this information Joyce enables an exploratory model of undo/redo; the user can navigate, visualize, edit and experiment with the history of the system safe in the knowledge that any history change will not have unforeseen and irreversible effects.

Introduction

Desktop applications have evolved into monolithic 'silos' of commands, each aware of the other only through relatively crude mechanisms such as cut/copy/paste. Today however, few tasks involve the use of just one application driven by one user; distributed, multi-application, multi-user, even multi-device scenarios are becoming the norm, but the interdependencies that arise are not made explicit and there is little recourse to repair errors.

This is most strikingly illustrated when one considers undo/redo systems. Most common undo/redo mechanisms fail to provide a satisfactory user experience because different applications (or even different sessions of the same application) are unaware of each other's behaviour, and because the user can only undo operations in reverse temporal order. To undo an operation made six actions ago one is forced to undo the five operations that follow it. Most applications also discard undo information for a session when the session is ended.

In contrast, we have designed a system-wide undo/redo facility called *Joyce*. Joyce uses a model of the system history that captures the *semantic relationships* between modifications rather than simply their chronological order. Explicitly modelling relationships allows us to more precisely articulate the *intentions* of applications and users and preserve these intentions when performing operations such as undo/redo.

Chronology is preserved in Joyce in order to apply history edits via a rewind/roll-forward mechanism and as a convenience to the user; any change to the operational history however, is evaluated using the relationship model.

Logging

Joyce runs as a system-level service that applications use to log modifications to their *artefacts*. For the purposes of this paper, we define an artefact to be the data that is edited by an application, for example, the artefact of a word processor is a document.

Following the *command pattern* [Gamma 95], applications provide discrete *actions* that reify their operations. Additionally, Joyce applications also formulate the *constraints* between their actions. A constraint is an object that represents some semantic relation; it is the responsibility of the system to maintain the constraint invariants. Joyce uses the full set of constraints defined by the IceCube reconciler [Preguiça 03b]; for the purposes of undo/redo the most important ones are *commutativity, atomicity* (indicating an all-or-nothing relationship between actions) and *causal dependency* (indicating that one action has caused another and the latter succeeding depends on the former succeeding).

Each user modification is evaluated by the application into a set of actions and constraints which are communicated to the Joyce service via IPC. As Joyce reads in the actions and constraints it generates an internal graph structure with actions as nodes and constraints as edges. This graph models the history of activity across the system in such a way that relations between operations are preserved.

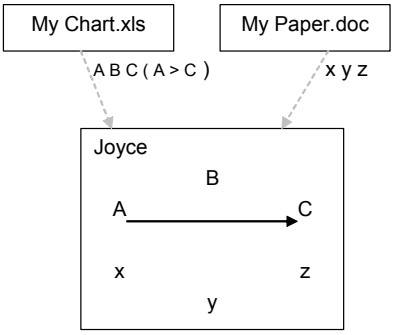

Figure 1: In this figure My Chart.xls has logged three actions, A, B, C and a causal ordering constraint indicating that C was caused by A. My Paper.doc has logged actions x, y, z

The action/constraint data recorded by Joyce is stored persistently, it does not disappear when an application is exited or a document closed. If the user is performing a complex task with many applications, he can express task-level dependencies by explicitly placing edges (i.e. constraints) in the action/constraint graph.

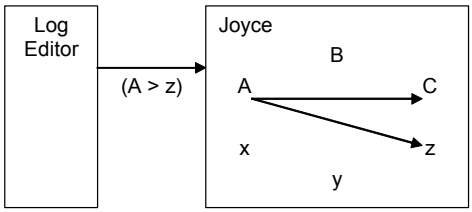

Figure 2: In this figure the user uses a log editor to explicitly place an ordering constraint between actions from different artefacts.

Milestoning and Rollback

Joyce applies system-level history edits at the application level by 'rewinding' artefact state and 'replaying' previously recorded actions. To facilitate this, applications using Joyce support *milestoning* and *rollback*. A milestone is a logical checkpoint that records the current state of the artefact at some point in its log. Sometime later Joyce may instruct the application to return the artefact to that state; this is called rolling back.

Milestoning is implemented either by creating a binary *snapshot* of an artefact for each successive milestone, or by ensuring that all actions applied to an artefact have corresponding *compensation* actions that nullify their effect. Rolling back to a milestone is then a case of either restoring the milestone's snapshot, or issuing compensation actions for every action chronologically after the milestone.

Log Editing and Edit Scope

We use the term 'log editing' to describe modifying the action/constraint graph explicitly (as opposed to generating it through regular application use). Undo and redo are log edits, as is placing a constraint between actions from different artefacts. The user may also *fix* an action by replacing it in the graph.

Log edits are made via a system-provided UI called the *log browser* that provides a visualization of Joyce's action/constraint graph. By default, the log browser shows a chronological view of the graph: actions are grouped with other actions from the same artefact and are displayed as a timeline. However, the user may also view the graph from different orientations. For example, to better visualize a prospective undo he may orient the graph to highlight causal dependency.

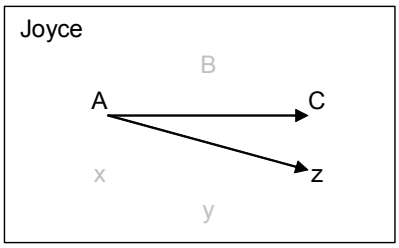

Figure 3: When the user selects A to be the subject of a log edit Joyce calculates the transitive closure of the dependant actions of A

Joyce provides a *selective* undo/redo/fix facility, meaning that the effects of an undo/redo/fix propagate only to those actions it needs to (contrast the stack-like, reverse temporal limitations of common systems.) This is achieved by using the action/constraint graph to compute the *transitive closure* of the actions dependant on the subject of the edit (e.g. the action to be undone). This transitive closure is termed the 'edit scope' and the actions within it are the actions that will be affected by the edit.

Note that since the scope is calculated using Joyce's internal graph it can easily span multiple artefacts. These are the artefacts that will have to be rewound when the edit occurs.

System-wide undo/redo/fix must take into account safety and security issues that do not usually concern application-local mechanisms. If Joyce decides it would be unsafe to undo or redo any of the actions in the edit scope the edit is denied. This includes the following cases:

1. An action in the scope is explicitly designated as not undoable or redoable. This would

include actions that cannot physically be undone or redone. Assume for instance an action that prints and posts a cheque and does not have any corresponding compensation action.

2. The log edit fails at some point in the scope. For instance, replacing an action during a fix may result in actions dependant on the fixed action failing dynamically.
3. The user does not have authorization to replay or undo an action in the scope. For instance an undo may involve modifying a file for which the user has no write permission.
4. An action that is not causally dependent would have to be undone or redone. An example is when some action in the scope is part of a transaction containing another action not in the scope. This restriction ensures that there is no "domino effect" [Randell 75].

Undo/Redo

Undo is the act of deriving the state the artefacts would have been in had the actions in the edit scope never happened. There are two alternatives approaches to do this:

1. Apply, in reverse causal order, a compensation action for each action in the edit scope.
2. Reset the artefact to the nearest snapshot prior to the chronologically first action in the edit scope. Then reapply the actions that occurred after this milestone *without the actions in the edit scope*.

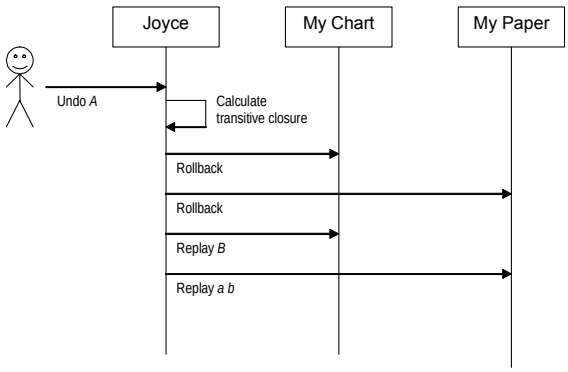

Figure 4: The user elects to undo *A* given the graph in figure 3. Joyce will calculate the edit scope, rewind the affected artefacts and replay the appropriate actions.

Joyce supports both methods and will decide which one to use for each artefact. If the application owning the artefact supports both methods, Joyce will use the most efficient in that particular circumstance. Additionally, Joyce takes advantage of the *commutativity* between actions to optimize undo and replay.

Redo is the inverse of undo; instead of extracting the edit scope from the constraint graph we re-insert it and replay from an appropriate milestone.

Fix is another log editing operation closely related to undo/redo. Fix is the act of replacing an action and having the effects of that change propagate to the dependant actions. Fix is similar to undo in that we roll back the actions in the edit scope. However, instead of removing the edit scope from the logs we re-run all the scoped actions to have them reflect the changed initial action. As noted earlier, this may result in failures that did not happen when the actions were first executed; in this case, fix is denied and every artefact retains its state as if the fix never happened.

Speculative and Comparative Undo

No information is thrown away when undo is performed: the actions in the edit scope remain in the action/constraint graph created by Joyce but are designated as not for execution. If the actions are redone they are simply marked for execution again. Since no information is lost we can provide a *speculative* undo facility that shows what the result of an undo *would* be. In this scenario, Joyce applies the undo in the normal manner in order to have it reflected in the artefacts. Once the undo has been applied Joyce prompts the user to accept or reject the undo. If the undo is rejected Joyce immediately applies the appropriate redo.

A natural extension of this is to allow the user to 'flip' between many speculative states. To achieve this, the user sets a milestone at the point he wants to 'fork' his artefact(s). He then generates a chain of actions representing one speculative state, rolls back to the fork checkpoint and generates a new chain of actions representing another speculative state etc. Joyce flips between states by rolling back to the fork milestone and rolling forward over one of the action chains. Once a user has settled on a state we simply continue appending actions from that point.

A similar effect is achieved in the Timewarp system [Edwards 97] by having the user fork into parallel timelines. Our method, in contrast, expresses the fork using the same action/constraint constructs that describe all system modifications and therefore no special logic is needed.

Structure of a Joyce Application

Most current application design is based around the Model-View-Controller [Gamma 95] pattern or one of its derivatives. Joyce extends this model by introducing a *coordinator* component. The coordinator is the bridge to the Joyce sub-system and is responsible for ensuring that the application model reflects the user interaction and the system-level log edits (such as undo/redo.)

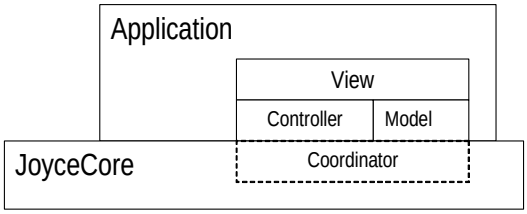

Figure 5: The abstract structure of an application using JoyceCore

To minimize application changes we provide a .NET assembly, *JoyceCore*, which handles much of the interaction with the Joyce service. Applications using the assembly provide implementations of the model, view and controller components and use a coordinator component provided by the library. This component provides:

- *Logging:* The application-supplied controller notifies the JoyceCore coordinator of the actions/constraints corresponding to an interaction.
- *Snapshots:* The JoyceCore coordinator provides a snapshot facility using .NET binary serialization to 'pickle' the application's model component.
- *Log editing:* The library will handle restoring an appropriate milestone and calculating the actions to replay after a log edit. The application needs to be capable of playing an action when supplied with the action instance.

This arrangement allows application developers to continue to concentrate on the MVC pattern and we can more easily integrate with existing frameworks.

Related work

Most of the common forms of undo/redo derive from the command object pattern [Gamma 95]. In this pattern the application records a chronologically ordered *history list* of objects that encapsulate modifications. Undo/Redo works by traversing this list backwards and forwards one command at a time.

There has been a lot of work to enrich this linear, chronological model. Chimera [Kurlander 88] provides graphical representations of past states to enable the user to better navigate his history. Timewarp [Edwards 97] extends the command pattern by allowing diverging or 'forking' timelines and by making timelines a first class object in the application's interface.

These systems, although richer than the standard command pattern, essentially try to solve the undo/redo problem by modelling *time* in the application. Time however, does not adequately capture the causal dependencies of modifications; only the application and the user know what actually depends on what. Flatland [Edwards 00] addressed this by grouping causally related commands into one unit that is removed/re-appended to a timeline atomically. However, this model assumes that causal actions are chronologically contiguous (they must immediately follow each other in the timeline) and does not allow the user to explicitly define causality.

Perhaps most closely related to our work is "Undo for Operators" [Brown 03]. The author presents a general-purpose undo/redo system intended as a tool for system administrators to repair configuration errors. The whole system is rolled back to the snapshot before the mistaken action; the fix is made; and subsequent actions are replayed. However, "Undo for Operators" does not have Joyce's edit scope concept, which is essential for security, reliability and exploratory undo and does not enable the user to articulate task-level dependencies. Furthermore, Joyce better integrates with interactive applications by intercepting the MVC loop.

Joyce is a successor to our earlier IceCube system [Preguiça 03a]. IceCube is a reconciliation system wherein reconciliation is driven by application semantics reified in the form of constraints. Whereas IceCube was a proof-of-concept prototype, Joyce is a full-featured distributed system designed to support real applications. The undo/redo/fix facilities that are the focus of this paper constitute a special case of Joyce's replication mechanism.

Concluding Remarks

Decoupling actions from their originating applications has clear advantages for the user experience and provides a better architecture for today's systems.

By using a system of actions and constraints we can construct a model of application history that

spans the whole system and persists as long as the user requires it. The information that this model captures can be used to provide a system-wide undo/redo/fix utility that is intelligent enough to constrain the effects of a history edit to only the dependent actions. This has two major benefits: firstly the user can make a history edit *anywhere* (as opposed to the last-in-first-out structure of current systems) and secondly the user knows that the history edit will not have unforeseeable adverse effects. These advantages combine to make undo/redo using Joyce explorative rather than punitive. Undo/redo/fix becomes less a tool for correcting errors and more a tool for asking *what if?*

Moreover, generating an action/constraint graph at the system level can also alleviate the phenomenon of having monolithic applications perform many jobs. For example, Microsoft Outlook rolls email, calendaring, contact management and to-do lists into one application. The Joyce system facilitates smaller apps exporting specialized actions that can be weaved together using constraints.

Joyce is currently written to the .NET platform and is undergoing testing and integration with a variety of applications.

References

[Brown 03] Brown, Aaron B., Patterson, Davis A., *Undo for Operators: Building an Undoable E-mail Store*. USENIX Annual Technical Conference, June 2003

[Edwards 00] Edwards, Keith W., Igarashi, T., LaMarca. A., Mynatt, Elizabeth D., *A Temporal Model for Multi-level Undo and Redo*, In Proceedings of the 13th annual ACM symposium on User interface software and technology, p.31-40, November 06-08, 2000

[Edwards 97] Edwards, Kieth W., Mynatt, Elizabeth D., *Timewarp: techniques for autonomous collaboration*, In Proceedings of SIGCHI conference on Human factors in computing systems, 1997, pp. 218 - 225

[Gamma 95] Gamma, E., Helm, R.., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts. Addison-Wesley, 1995

[Kurlander 88] Kurlander, D., Feiner, S. *Editable graphical histories*. In IEEE Workshop on Visual Languages (1988).

[Preguiça 03a] Preguiça, N., Shapiro, M., Matheson, C., *Efficient semantics-aware reconciliation for optimistic write sharing*. In Proc. Tenth Int. Conf. on Cooperative Information Systems (CoopIS), Catania, Sicily, Italy, Nov. 2003.

[Preguiça 03b] Preguiça, N., Shapiro, Martins, Legatheaux J., SqlIceCube: *Automatic Semantics-based Reconciliation for Mobile Databases*, Technical Report 2-2003 DI-FCT-UNL, Universadade Nova de Lisboa, 2003

[Randell 75] Randell. B., *System structure for software fault tolerance*. IEEE Trans. on Software Engineering SE-1, 2 (June 1975), 220-232.]