Using Task Models for Cascading Selective Undo

Aaron G. Cass and Chris S. T. Fernandes

Union College, Schenectady, NY 12308 USA,
{cassa, fernandc}@union.edu

Abstract. Many studies have shown that selective undo, a variant of the widely-implemented linear undo, has many advantages over the prevailing model. In this paper, we define a task model for implementing selective undo in the face of dependencies that may exist between the undone action and other subsequent user actions. Our model accounts for these dependencies by identifying other actions besides the undone one that should also be undone to keep the application in a stable state. Our approach, which we call *cascading selective undo*, is built upon a process-programming language originally designed in the software engineering community. The result is a formal analytical framework by which the semantics of selective undo can be represented separately from the application itself. We present our task model, the selective undo algorithm, and discuss extensions that account for differing kinds of inter-action dependencies.

1 Introduction

Most applications that support an undo command use a *linear undo* model. Under this model, only the most recent user action is retracted, and multiple consecutive executions of the undo command iterate backwards through the history list. In general, given user actions $A_1, ..., A_n$, issuing an undo command will only undo action A_n, and one cannot undo any action A_i without also undoing actions $A_{i+1}, ..., A_n$.

However, the *selective undo* model, introduced by Berlage [3] and studied more formally by Myers and Kosbie [7], has long been studied as an alternative to the linear model. In selective undo, a user can undo an arbitrary action A_i without undoing other actions. This has significant advantages over the linear model. First, collaborative environments necessitate the implementation of a non-linear model. If multiple users are editing the same shared document over a network simultaneously, then any particular user's most recent *local* action may not be the most recent *global* action in the system. Therefore, any undo implementation must be selective by nature.

Second, selective undo encourages a user's exploration of an application, especially when the application allows for tasks to be done in an arbitrary order. Consider a word processor user who wants to change a document from a current state $\mathcal{A}$ to another state $\mathcal{B}$, but is unsure of the exact steps she should take to accomplish this. It would be preferable for the user to be able to perform a series of tentative steps from $\mathcal{A}$ towards $\mathcal{B}$ all the while knowing that she could return to state $\mathcal{A}$ at any time via undo – even if other unrelated tasks, such as the changing of a word at the behest of the spell checker, were performed during the exploration process. Selectively undoing the tentative steps without changing the results of the spell checking allows for greater flexibility than the linear model.

One of the biggest challenges in the implementation of selective undo, however, is in defining what should happen to subsequent user actions in the history list that are semantically dependent on the selectively undone item. Consider the following word processor commands:

1. Type “hello”.
2. Italicize “hello”.
3. Copy “hello”.
4. Paste in position x.

Selectively undoing the second action could result in the removal of italics from either the original and the pasted text or just the original text, depending on how we define the pasted text’s relationship to the original. Indeed, the repercussions of not dealing with dependencies can be more severe than just an unexpected change in formatting. In general, the selective undoing of a user action that creates an object A will result in ambiguity over how to interpret a subsequent user action in the history list that affects the value of some property of A. Dependencies such as these must be considered for a selective undo mechanism to handle complex tasks. In previous work, these dependencies are accounted for in limited ways, such as treating some tightly dependent user actions as null operations during undo [5] or disallowing the undo action if the result will not be meaningful [3]. We propose a different alternative – allowing an undone action to cause the undoing of other user actions until a meaningful state is reached (with appropriate user feedback and override controls). We believe this *cascading selective undo* offers more flexibility than previous approaches.

In this paper, we introduce a paradigm for cascading selective undo that explicitly models and tracks dependencies that exist between user actions. We model these dependencies using a process language originally designed for the software development community. We believe this to be a good match since software engineering is a domain in which dependencies between user tasks frequently occur, and thus provides fertile ground for determining appropriate undo semantics. In addition, tools already exist in the software engineering community for capturing dependencies in complex process or workflow models. Conversely, the tools which are used for software development can greatly benefit from a feature allowing selective undo.

Section 2 discusses related work in the area of selective undo. Section 3 explains our approach, detailing why we believe structure should be placed on user tasks, outlining motivating examples, and presenting our algorithm for cascading selective undo. Section 4 highlights future extensions to this approach, and Section 5 gives concluding remarks.

2 Related Work

The semantics as to how selective undo should work have not converged in the literature. Many implementations use the *script* paradigm [2], in which the result of undoing action A_i alone is equivalent to the result reached by executing user actions:

$$A_1, ..., A_{i-1}, A_{i+1}, ..., A_n$$

in that order. That is, if the list of user actions is viewed as a script, undoing one action is equivalent to removing that action from the script, with no other changes. This can result in side effects the user may or may not have intended, such as the changing of the pasted text's formatting in the example above. On the other hand, selective undo as discussed in the GINA system [3] simply restores the values of an undone item's properties to what they were immediately before the original execution of the user action. For our example above, this results in the removal of italics from just the original text. This ambiguity in the definition of selective undo points to the disparity between the way undo is perceived by interface programmers compared to users [1, 6]. Relationships between actions are often known to users of an application because they know how they are trying to use the system, but the system does not know the relationships and therefore cannot act on them.

Myers and Kosbie [7] later adopted the selective undo semantics used in GINA when they implemented their Amulet user interface. By implementing *command objects*, they organized user actions into a hierarchy which allowed higher-level commands to be invoked by lower-level commands. Their system not only supported selective undo, but selective reusability of arbitrary commands on new objects. However, since Amulet supported GINA semantics, their support for dependencies associated with selective undo is limited to undoing the values of command objects' properties. If a user used two command objects A and B where the output of using A was piped as input into B, the selective undoing of A may leave B in a semantically unstable state.

Regarding undo in collaborative frameworks, Abowd and Dix [1] point out that even *defining* undo from a user's perspective can be a daunting task. Chen and Sun [4] use the script model to implement their Any Undo feature (described in [11]) in the GRACE collaborative graphical editing system. Prakash and Knister [9] augment their DistEdit package–a group text editor building kit–with a form of selective undo where a *Conflict* function is used to check to see if two operations are dependent on each other in such a way that one cannot be undone with the other also being undone. And Ressel and Gunzenhäuser [10] use dynamic transformation rules to effect undo in a collaborative system by allowing transformations to affect future transformations in a pre-determined way. However, in all of these systems, any accounting of dependencies is done through the implicit encoding of them into the undo algorithms themselves. A major novel aspect of the model we propose is that dependencies and undo semantics are abstracted to an external representation. This allows the semantics of undo to be separately represented apart from the application code itself.

3 Our Approach

Before detailing a couple of examples that will be used to explain our approach, let us first explain that the approach assumes that an application imposes structure on how a particular user interacts with it. We argue (a) that imposing some structure on the use of technology can have distinct benefits for supporting users and (b) that many applications already do this. Consider, for example, automated teller machines (ATMs). Most ATMs ask the user to enter a personal identification number (PIN) before any other action. The underlying reason to request a PIN is that the user must be authenticated

before the system withdraws money from their bank account. However, the system need not authenticate the user as the first action – it can wait to get authentication when it is directly needed. However, there are good (usability and security) reasons to request the PIN first. Therefore, imposing structure has benefits and is currently done in applications of many sorts.

Of course, not all structures are good for all uses or all users. A task structure designed to help one use *PowerPoint* to create an organization chart will clearly not help one create a slide presentation. So, instead of forcing one structure, we propose that multiple structures be made available, thus creating different *applications* that use the underlying application as a basic technology – in essence, the user interacts with the composition of the underlying application and a structure that specifies a *way of using the application*. We also propose that these models be relatively high-level, thus allowing the user to flexibly use the low-level application for low-level tasks but guiding them to string the low-level tasks together to solve the higher-level objectives.

Note that users of current applications have (at least a rough) understanding of the way they wish to use their applications. Unfortunately, the user does not have a way to indicate to an application how the application will be used and therefore cannot get specific guidance from the application itself. We propose that users can choose the way they want to use a system by choosing a particular task model, thus informing the system of their choice. The system can therefore use the chosen task model to constrain use of the system, thus guiding the user in their task.

Note also that the task models are not created by the end users of the system. While end users can certainly be involved in the specification of ways of using applications, we imagine that these models will be generated by domain experts. These experts will create specific task models that they expect with be of value, either because they describe very common tasks, popular uses of the underlying application, or uses needed within a particular organization.

The task models can be arbitrarily flexible, effectively removing all constraint on use. So, the approach we describe does not require imposing arbitrary structure – but we expect that when structure is imposed, we can give the user benefits. In particular, knowing the structure of the task the user is trying to accomplish can help us provide what we believe is a more natural mechanism for the *undo* command that takes this structure into account. It is the meaning of undo in these structured situations that we discuss in this paper. We start with an explanation of two examples that illustrate the desired behavior of undo in different scenarios.

3.1 Motivating Examples

Consider, as a running example, a scenario in which two authors are using a presentation application, such as *PowerPoint*, to create a single presentation, and they wish to impose a structure upon their presentation in the form of sections, subsections, etc. with the slide being the most basic unit of a section. They wish to have a Table of Contents slide at the beginning of the presentation that shows a list of the section names. This slide is then repeated at the beginning of each section with the next immediate section name highlighted.

We consider two examples to demonstrate the mechanics of cascading selective undo.

Example 1. Consider the following user actions:

1. User 1 begins the process of adding a section by creating a section header.
2. User 2 updates the Table of Contents slide to include the new section.

Suppose User 1 then decides against the creation of the new section and undoes step 1. Since the Table of Contents slide content is dependent on the existence of sections, the update in step 2 should also be undone. In other words, the undo should cascade to step 2. It is this dependency that must be represented. At first glance, this action is equivalent to what would happen under the linear undo model, but the second example shows how selective undo reacts in a more complex situation.

Example 2. Consider the following set of user actions. To save space, we have grouped multiple user actions into single steps in places where the point of the example is not affected:

1. User 1 creates slides A_1 through A_n.
2. User 1 creates section title slides S_1 through S_m.
3. User 1 creates the Table of Contents slide.
4. For each slide A_i, User 1 places it into its appropriate section, S_j.
5. User 2 creates slides A_{n+1} through A_{n+k}.

Suppose that after User 2 creates the last k slides, User 1 realizes that the section structure is not appropriate and wishes to delete it, perhaps because the new content does not fit in the existing structure. Selectively undoing step 2 would accomplish this. Intuitively, we want our undo mechanism to be aware that the presentation structure is independent of the slides, so that undoing step 2 would cascade to steps 3 and 4 but not to step 5. While this accomplishes the same goal as deleting the structure manually, undo is more efficient because the structure, with all of its nested subsections, may be spread throughout the presentation, making it difficult for the user to find and delete all occurrences by hand.

Both of these examples show how cascading could be used to bring the user document back to a stable state. We now describe the language and algorithms that we have developed to accomplish this.

3.2 Modeling Dependencies

Any number of formalisms could adequately model control dependencies between tasks. However, some software engineering researchers have developed *process-programming* languages [8], which not only model task dependencies, but also enable automated execution of these task models to track user progress on the tasks. We chose to use Little-JIL [13], a recently-developed, feature-rich, graphical process-programming language. One of the authors (Cass) has been involved with the development of this language and the infrastructure that supports it, and has used it to develop a software design tool. We now show how this language can be used in the context of the previously-described examples.

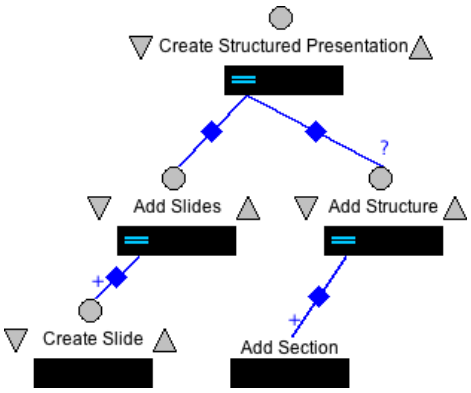

Fig. 1. A Little-JIL program that shows the overall process, elaborated further in other figures, for creating a structured presentation.

Figures 1, 2, and 3 show a task model[1] for the application presented in the previous section. The task model represents the overall task as an elaborated hierarchy of tasks, while the *kind* of a parent task determines the order of execution of its child tasks. The Create Structured Presentation task in Figure 1 is of the *parallel* kind, as indicated by the two horizontal bars. Because it is a parallel task, its children, Add Slides and Add Structure, can be performed in any order, to be chosen by the users of the system. In essence, we are saying here that these tasks do not depend on one another for the reasons outlined in the previous section.

Figure 1 also shows the cardinality mechanism of the language. The question mark on the edge above Add Structure indicates that this task is optional in this context. The users of the system need not add any structure to the presentation. However, if they decide to add structure, they must add one or more sections, as indicated by the plus sign on the edge above Add Section. Note that Add Structure is parallel, indicating that the sub-tasks involved in adding sections do not depend on each other – sections are independent.

Note also the two tasks Create Slide and Add Section. Create Slide is a leaf task, a primitive task actually performed by a user, as indicated by the lack of a task kind. Add Section on the other hand, is a *reference*, as indicated by lack of the triangles on either side of the task name[2], and is therefore further elaborated somewhere else.

In this case Add Section is elaborated in Figure 2. As indicated by the arrow, Add Section is a *sequential* task and thus its child tasks must be performed in left-to-right order. The user must add a section header first, update the corresponding table of contents slides, and then elaborate the section. What is not shown in the diagram is that not all of these tasks need be performed by the same user. In fact, it is relatively easy

[1] We use the word *task model* to emphasize that we do not specifically require a process-programming language. In the software engineering literature, task models are called *process programs* and tasks are called *steps*.

[2] These triangles represent pre- and post-requisites, which are not used in this example.

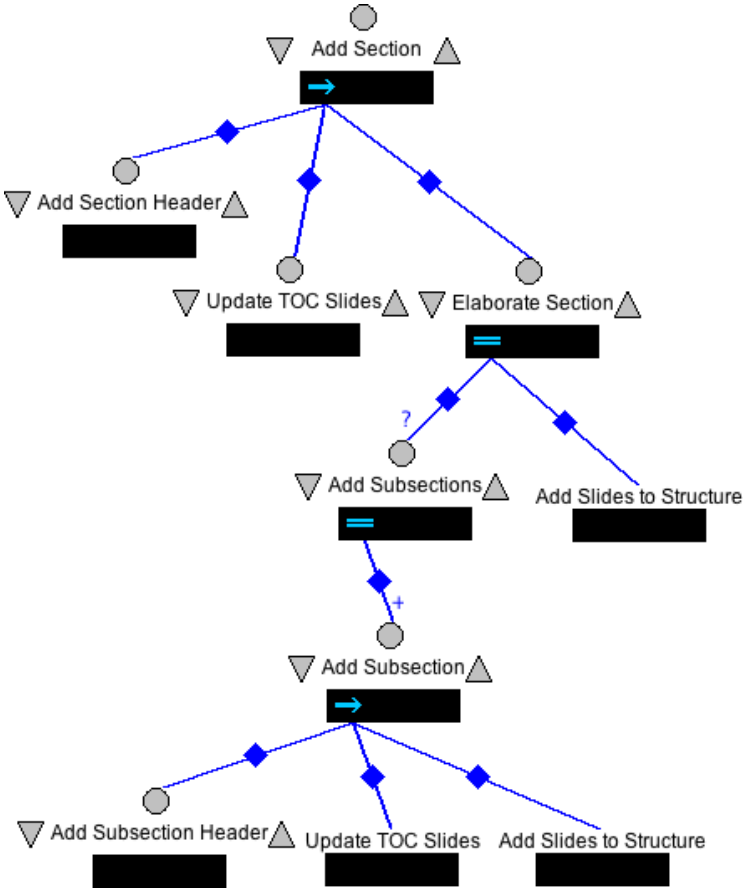

Fig. 2. An elaboration of the Add Section step from Figure 1.

to arrange for some of the tasks to be automated – for example, the system could automatically update the table of contents pages because such pages are fully determined by the section and subsection headers added by users.

Note that this particular way of creating presentations is not novel, and in fact there are applications that directly support this method (for example, the Beamer [12] LaTeX package). However, we aim for a more flexible, general approach that would support multiple ways of using the same underlying tools to perform different tasks – by encoding the different ways of using the tools in task models, the system can support and guide the users to use the tools in different, useful ways.

Note that there are other task kinds not used in our example (see Table 3.2). By changing the task kinds, we can easily create different user experiences. For example, changing the kind of the Create Structured Presentation task to *sequential* indicates that one should create slides first and then add structure. For novice users, one might want to

Table 1. The different task kinds supported by Little-JIL.

Kind	Symbol	Description
Sequential		Child tasks must be performed in left-to-right order.
Parallel		Child tasks can be performed in any order, possibly overlapping.
Choice		One child task, chosen by the users, must be performed.
Try		Child tasks are tried in a specified order; task is done when one child succeeds.

give more precise guidance, while for experts, one might want more choice and parallel tasks.

Continuing with the example, Elaborate Section involves (optionally) adding subsections and adding slides to the section (if we did not want to add slides to the section, we would not have added the section). Add Subsections is defined similarly to Add Structure. Both reference Add Slides to Structure, which is elaborated in Figure 3.

Though not shown, the language also supports a *parameter-passing* mechanism. For example, to enable Add Slides to Structure to be used in the two previously-described contexts, we can define it to take as a parameter the name of the section or subsection to which to add slides. Update TOC Slides would also take the section or subsection header as a parameter.

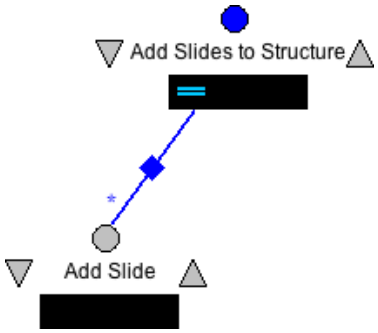

Fig. 3. An elaboration of the Add Slides to Structure step from Figure 2. The star (*) cardinality indicates that zero or more slides may be added by the user.

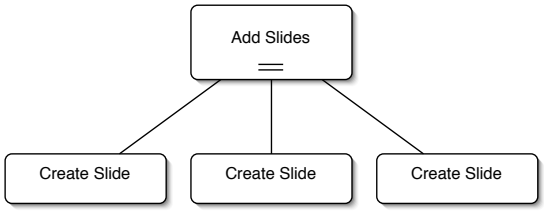

Fig. 4. A possible instance tree for the Add Slides task.

3.3 The Main Cascade Algorithm

Given the dependencies modeled in a task model of the kind outlined above, and given a user request to undo one of the tasks, a cascading selective undo algorithm must calculate the *cascade*, i.e. the set of tasks that must also be undone if the requested task is to be undone. The algorithm must ensure that if only the selected task and the tasks in its cascade are undone, the system will arrive at a meaningful state. In other words, the algorithm must ensure that the system after the undo command is completed is one that could have been reached by following the task model on a path from the starting state.

To clarify the context in which such algorithms must work, let us note here that the task model is *instantiated* as the users interact with the system. The tasks in the task model represent *types* of tasks that are instantiated in different contexts. For example, Elaborate Section is instantiated with different parameters. As another example, consider the *instance tree* shown in Figure 4, which is one possible instantiation of the Add Slides sub-tree from the task model. In this example, there are three instances of Create Slide, indicating that users have created three slides using the system. Note that the instances are different. In particular, the first instance of Create Slide is different from the second instance – the second instance is optional because the first one will have satisfied the cardinality of the task. Both instances share the same parent instance (an instance of Add Slides), but they differ in this important context. The task model is therefore a model of task types, while the algorithm must work with task instances.

Simpler task-modeling formalisms that model task instances directly afford relatively simple cascade logic – if a dependency edge exists between the task instances, the target task is in the cascade of the source task. However, such simpler task-modeling formalisms would not allow reuse of tasks in different contexts, would not model naturally-hierarchical tasks, and would not allow specification of cardinality. Our algorithm, on the other hand, must compute the dependency relationship between pairs of task instances based on relationships between task types. The input, therefore, to our algorithm is a task instance to be undone, while the output is the set of all task instances that must also be undone to cause the system to arrive at a state consistent with the task model. The main algorithm, which deals only with control dependencies specified by the task kinds, follows:

```
COMPUTECASCADE(t)
Input: Task instance t to be undone.
Output: Cascade for t.
 1: C ← {t}
 2: if t is root task then
 3:     return C
 4: p ← t.parent
 5: k ← p.kind
 6: C ← C ∪ COMPUTECASCADE(p)
 7: if k = SEQUENTIAL ∨ k = TRY then
 8:     C ← C ∪ subtrees(p.childrenAfter(t))
 9: return C
```

The algorithm, as outlined above, assumes that task instances have the following attributes:

- *parent*: The parent of task instance.
- *kind*: The task kind of the associated task.

In addition to these attributes, task instances need a *childrenAfter* function, which returns the children of the task that are to the right of the given child – that must come after the given child in the sequential ordering of the sub-tasks. We also use a *subtrees* function to find all tasks in the subtrees rooted at those children.

Theorem 1 COMPUTECASCADE*(t) correctly computes the set C of all existing tasks dependent on t by a control dependency specified in the task model.*

Proof. Consider a task s dependent on t by a control dependency in the task model. Because the only dependencies we consider are those specified by the task kinds, s depends on t because they share a common ancestor, $\mathcal{A}$, that makes that dependency explicit. Parallel and choice kinds do not control the order of execution of their sub-trees, so $\mathcal{A}$ is neither parallel nor choice – it must be of sequential or try kind. We know that s is not in a sub-tree to the left of t's sub-tree at $\mathcal{A}$ – this would imply that s *necessarily* had to be performed before t, not the other way around. Therefore, s is in a sub-tree to the right of t's sub-tree at $\mathcal{A}$. Because the recursion starts at t and moves up the tree, the algorithm eventually reaches a child of $\mathcal{A}$ and then line 8 of the algorithm adds s to the set C.

Similar logic can be used to show that no task is added to C if it is *not* dependent on t. Therefore, we have shown that the set C is equivalent to the set of tasks dependent on t.

Let us now return to the motivating examples to see how the above algorithms would produce the desired results. Figure 2 shows the tasks relevant to Example 1: namely Add Section Header and Update TOC Slides. When the Add Section Header task is undone, the sequential control dependency of its parent causes line 8 of the main cascade algorithm to execute. This causes all children of the sequential node Add Section which are to the right of Add Section Header to also be undone, including the update of the table

of contents. Note that if the Elaborate Section task had also been completed by the user (to include subsections, for example) then that task would have been undone as well.

Example 2 involves a larger part of the task model. Steps 1 and 5 of the example, dealing with the creation of slides, is handled by the subtree rooted at the Add Slides node in Figure 1. Steps 2-4, dealing with structure, is handled by the subtree rooted at the Add Structure node in the same figure, whose child node Add Section is fully modeled in Figure 2. In this example, undoing the creation of all of the section title slides is equivalent to undoing *all* of the Add Section tasks. In other words, it is equivalent to undoing the parent node, Add Structure. When the algorithm is applied, all nodes on the path from the Add Structure node to the root of the task model, which in this case is just the Create Structured Presentation node, will also be undone. However, note that because the Add Slides task has an explicit parallel dependency relationship with Add Structure, none of the slide creation steps will be undone, which is exactly the desired result for steps 1 and 5 of Example 2. The final step is to observe that, unlike the previous example, the user wishes to undo a non-leaf node. For that case, we also wish to undo all leaf node tasks that are descendants of Add Structure. This will cause the individual user steps 2, 3, and 4 of Example 2 also to be undone, as desired. Note that this requires additional work beyond computing the cascade – we consider descendant tasks separately from the cascade.

3.4 Extensions to the Approach

The algorithm presented above details proper semantics for cascading selective undo for cases in which nominal control flow defines all the dependencies between tasks. However, the approach can be extended to take advantage of cardinality specifications and data dependencies.

Cardinality Open-ended cardinalities open up an interesting possibility for selective undo. Consider, for example, undoing a task instance that is attached to an edge with + cardinality, such as one of the Create Slide instances shown in Figure 4. If the user wants to undo all tasks instances attached to the edge, it is clear that the parent must also be undone because the cardinality specification requires at least one sub-task. However, if we are undoing only some of the tasks, leaving at least one completed task attached to the edge, the cardinality can be satisfied by the remaining tasks. In the case where the parent is a parallel task, we can safely avoid further cascading (aside from the recursive searching for sequential and try ancestors) because the siblings do not depend on one another. The algorithm for this approach is similar to the one above, except for the special case noted here.

Note, however, that this extension would not always produce the result the user wanted – if the user wants to undo a task so that they can redo it in a different way or with different information, then we require that the parent task be undone, because in order for tasks to be performed, their parent tasks must not have been completed. We therefore plan to investigate ways of giving the user the choice of these two interpretations – perhaps presenting both cascades.

Data Dependencies In some situations, data dependencies between tasks can be known *a priori*. For example, a task that modifies a slide clearly depends on the task that created the slide. In the current system, we model some of these data dependencies using the parameter-passing mechanism, which specifies parameter passing between parent and child tasks. If a task $\mathcal{B}$ depends on data from a sibling task $\mathcal{A}$, we indicate this by passing a parameter from $\mathcal{A}$ to its parent and subsequent passing of the same parameter from this parent to $\mathcal{B}$. Note that this is not limited to direct siblings – we can specify data flow between any two tasks that share a common ancestor. However, this mechanism only allows specification of data dependencies between $\mathcal{A}$ and $\mathcal{B}$ if $\mathcal{A}$ and $\mathcal{B}$ are already related by a control dependency. In particular, if the common ancestor is a parallel task, the system cannot guarantee that the parameter received by $\mathcal{B}$ is the one produced by $\mathcal{A}$ – $\mathcal{A}$ might not yet have executed at the time that $\mathcal{B}$ starts. Therefore, the set of data dependencies specified in our task models is a subset of the control dependencies already specified.

In some situations, this is not enough to model all known data dependencies. For example, we might wish to have an Add Section Cross-Reference task as an optional child of Create Slide in our running example. This task might need the section header as a parameter from the Add Section Header task that created the section, which therefore means that Add Section Cross-Reference must happen *after* the corresponding Add Section Header task. In the current language, we cannot specify this data dependency without forcing sequential control dependency between the two tasks. Our plan is to add independent data dependencies to the language. Of course, when we have added these new dependencies, our undo algorithms will have to consider them in computing the proper cascade for a user's undo request. This should be possible by taking the union of the data dependency cascade with the one computed by the main algorithm.

Note that not all data dependencies can be known *a priori*. Consider a development environment for creating Java programs. At runtime, the user might create a method in a previously-created class, even though the task model did not require that the method be created in exactly this class. If the user requests that we undo the creation of the class, clearly the creation of the method must be part of the cascade. Because this data dependency is created by the user's actions and not required by *a priori* dependencies, the previously-outlined approach cannot capture it. Berlage [3] suggests that these kinds of dependencies should be modeled in the tool and that they can be used to disallow undo commands that result in "meaningless" states. We suggest taking this a step further by using these dependencies in calculating the proper cascade. Our new algorithms will have to support this.

4 Future Work

In addition to the extensions outlined above, the development of which we already have underway, we plan further improvements to the approach as well as development of visualizations to support users in using it. We further plan to run experiments to assess the usability of the approach.

Fig. 5. One possible visualization of a cascade. The user has selected "Create Section" to undo.

4.1 Dealing with Exceptions

Little-JIL [13] also has an exception handling mechanism, which allows the specification of scoped handlers that respond to exceptions thrown from tasks up the hierarchy. This presents some additional issues for selective undo. If a user action causes an exception to be thrown, then undoing that action should cause a cascade to the exception handler, which should also be undone. Because exception handlers are themselves represented as task hierarchies, consisting of an arbitrary number of subtasks with control dependencies between them, undoing an exception handler involves undoing those subtasks.

Cascading of undo operations to exception handlers is not currently handled by our system, but will be in future implementations. One interesting open question is if users should be allowed to *directly* undo exception handlers themselves, perhaps with the intent of redoing the handling in a different manner. This is under investigation.

4.2 Limiting Scope of Undo

There are scenarios in which an operation cannot be undone. For example, an application might have a finite history list or it might model a real-world permanency. (One cannot undo a transaction if the seller has already spent the money or the buyer has already consumed the good.) In these situations where a full cascade is not possible, the application should disallow undo. However, the undo logic may not realize that it is in this situation until a cascade is already in progress. In this case, we must be able to reverse the effects of the cascade. Alternatively, the cascade could throw an exception, to be handled as we discussed above. There are plans to address these issues in future versions.

4.3 Visualization for the Approach

An important aspect we have not yet discussed is how a cascading undo will be represented to the user, both before and during the cascade. When the user selects an action to be undone, it should be clear what the cascade will affect. In Example 1, one wouldn't want the user to be surprised when the Table of Contents slide is altered. One possible visualization would be to simply show the user all affected actions, including system actions. An example appears in Figure 5. Alternatively, we could couple a user's selection with an explanation of why other actions were being affected. Explicit control dependencies would allow such explanations to be created dynamically.

4.4 Task Model Inference

The approach described here assumes that the task model is known ahead of time. Several task models might exist for a particular application, but the user chooses one of those models when they begin using the system. Of course, this can be limiting. In the longer term, we plan to explore approaches that would infer the task model the user intends to use by monitoring low-level user actions. Then, instead of first choosing a task model, the user can explore with relatively more freedom early on in a session before giving enough context to the system that the system can then start to help the user make continued progress.

Of course, if the user can choose a task model at the beginning of use, they can also change task models mid-way through a session. Again, under the current approach, the user must make an explicit choice. However, we envision an approach that would monitor user actions and reinterpret the user's choice at those points when the user attempts tasks that are not in the currently "chosen" task model.

4.5 Evaluation of the Approach

There are several important questions to answer with respect to the usefulness of selective undo to end users. Some of these questions are:

1. Does our interpretation of selective undo semantics match user expectations?
2. Would a user choose to use selective undo over linear undo?
3. Would a user choose to use selective undo instead of performing an alternate set of tasks? (a more general case of the above question)

We have begun experiments to help answer these questions. Here we describe one completed experiment and two yet to be carried out.

In the first experiment, we have tried to determine whether a user would choose to apply selective undo instead of the traditional linear model. We provided subjects with a series of drawing tasks, and asked them to show the state the drawing should be in after one of the tasks is undone. We then coded their responses to infer which undo model each subject performed. In this experiment, we found that cascading undo is chosen more than the other two and linear undo is chosen extremely infrequently.

However, we cannot conclude from this experiment that undo is more desirable than performing an alternate set of tasks. To explore this question, we plan an experiment where we guide subjects to create and edit elements in a document to reach a certain state before asking each subject to transform the document to a different state. If this different state is well-chosen, the subjects can choose a selective undo to reach the state, but they can also use a different set of tasks (creation, editing, or deleting tasks) to reach the same visible state. We predict that subjects will choose selective undo over using a different sequence of tasks, except perhaps if the alternate sequence is very short.

In another experiment, we wish to determine if users can predict what will happen (i.e. the end state) if either linear or selective undo is used in a given context. We will give the user a series of tasks on a familiar application and then tell the user that either linear undo or selective undo will be used at a particular point. We will then ask the user to predict what the document will look like after that particular type of undo has been

executed. We expect to use a broad, computer-literate population for this experiment in order to determine if a particular paradigm for selective undo is the "natural" one.

5 Conclusions

We have presented cascading selective undo, which maintains the advantages over linear undo that have long been touted while also capturing dependencies between user tasks that are necessary in many contexts to ensure that undo results in a meaningful state. Our approach exploits task models designed by application developers and chosen by users – task models that we argue provide much needed guidance in complex domains and provide additional context information to the system when the user requests an undo action.

Our novel approach uses Little-JIL, a process modeling language from the software engineering community, to explicitly represent task models with control dependencies. Little-JIL provides precise representation of control dependencies that have allowed us to develop algorithms that determine appropriate cascading of undo at any point in the user experience. It is also robust enough to allow for complex task models where the semantics of selective undo are not straightforward. Extensions to our main algorithm account for the complexities in the model, including cardinality, exception handling, and data dependencies.

We are currently planning experiments to test the viability of selective undo to end users. Results will allow us to refine our model, the language upon which it is based, and implementations of selective undo. We already have a Little-JIL-based implementation of a software design tool that shows that the language is capable of representing complex executable task models. It does not yet support undo (selective or otherwise), but based on the knowledge gained from the current work, and the experiments outlined in the section on future work, we plan to add a selective undo mechanism to the tool in the near future, and thus learn whether the approach described here is feasible for developers as well as for users.

References

1. G. D. Abowd and A. J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.
2. J. E. Archer, Jr., R. Conway, and F. B. Schneider. User recovery and reversal in interactive systems. *ACM Transactions on Programming Languages and Systems*, 6(1):1–19, 1984.
3. T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Transactions on Computer-Human Interaction*, 1(3):269–294, 1994.
4. D. Chen and C. Sun. Undoing any operation in collaborative graphics editing systems. In *GROUP*, pages 197–206, 2001.
5. A. Dix. Moving between contexts. In P. Palanque and R. Bastide, editors, *Design, Specification and Verification of Interactive Systems '95*, pages 149–173. Springer, 1995. Toulouse, France.
6. R. Mancini, A. J. Dix, and S. Levialdi. Dealing with undo. In *Proc. of INTERACT'97*, Sydney, Australia, 1997. Chapman and Hall.

7. B. A. Myers and D. S. Kosbie. Reusable hierarchical command objects. In *Proc. of the ACM Conf. on Human Factors in Computing (CHI 96)*, pages 260–267. ACM Press, 1996.
8. L. J. Osterweil. Software processes are software, too. In *Proc. of the Ninth International Conf. on Software Engineering*, Mar. 1987. Monterey, CA.
9. A. Prakash and M. J. Knister. A framework for undoing actions in collaborative systems. *ACM Transactions on Computer-Human Interaction*, 1(4):295–330, Dec. 1994.
10. M. Ressel and R. Gunzenhäuser. Reducing the problems of group undo. In *GROUP*, pages 131–139, Phoenix AZ, USA, 1999.
11. C. Sun. Undo any operation at any time in group editors. In *Computer-Supported Cooperative Work (CSCW)*, pages 191–200, 2000.
12. T. Tantau. *User's Guide to the Beamer Class, Version 3.06*. http://latex-beamer.sourceforge.net, Oct 2005.
13. A. Wise, A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, and S. M. Sutton, Jr. Using Little-JIL to coordinate agents in software engineering. In *Proc. of the Automated Software Engineering Conf.*, Grenoble, France., Sept. 2000.