Modeling Dependencies for Cascading Selective Undo

Aaron G. Cass and Chris S. T. Fernandes

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

Abstract. Linear and selective undo mechanisms have been studied extensively. However, relatively little attention has been given to the semantics of selective undo in situations in which the undone command has dependency relationships with actions later in the command history. We propose a new approach to selective undo using a process modeling language from the software process community to model dependencies between commands, and we outline semantics for undo in the face of these dependencies.

1 Introduction

In most applications today that implement an undo feature, a history list is kept of all user actions, and a *linear undo* model is followed. In this model, an undo command will only reverse the most recent action, and subsequent undo commands will iterate backwards through the history list. For example, 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 $A_{i+1}, ..., A_n$.

Other undo paradigms exist, however, and many have studied non-linear undo models. One such model is the *selective undo* model, introduced in [1] and studied more formally in [2]. In this model, a user can undo action A_i without undoing other actions. The meaning of this kind of undo can differ, however. Most implementations of selective undo use the "script" paradigm [3], 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. If the given actions were the following word processor commands:

1. type "hello"
2. italicize "hello"
3. copy "hello"
4. paste in position x

then selectively undoing the second action would result in the removal of italics from both the original and the pasted text. On the other hand, the semantics of selective undo in [1] would result in the removal of italics from just the original text. This semantic ambiguity points to the disparity between the way undo is perceived by interface programmers compared to users [4, 5].

In this position paper, we explore the semantics of selective undo using ideas from the software process community. We believe this to be a good match since a primary cause of the ambiguity discussed above is the dependencies which can exist between user actions in any given "script". In the above example, this dependency is seen when the italicized text is duplicated. Software engineering is a domain in which these dependencies 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.

One novel aspect to our approach is the way in which we handle recognized dependencies between user sub-tasks. In previous work, these dependencies were either not accounted for at all or else handled by disallowing the undo action if the result was not meaningful, such as in [1]. An example of a selective undo that creates a meaningless result is if a programmer first created a class "Car" and second created a method "fillTank" for "Car". Selectively undoing the first action would cause the second action to be meaningless and under the semantics presented in [1] would be disallowed. 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 and that software engineering approaches can be used to help determine a meaningful cascade.

Section 2 provides details about the software engineering concepts we will employ to capture the context of selective undo. Section 3 gives examples of the types of questions we will explore. Section 4 examines future work and conclusions.

2 Our approach

Cass has been developing software design environments for use in design guidance experiments [6] using programs written in a process-programming language called Little-JIL [7, 8]. Such process programs allow for a wide range of guidance to users of a design tool, from very flexible to very constrained sequencing of tasks. A process program specifies the allowable sequences of tasks and the data flow between them. The Little-JIL run-time environment works with this process model to make commands available to users on a hierarchical to-do list. As the user performs tasks, the system presents new tasks as controlled by the semantics of the Little-JIL program.

The current implementation does not support undoing user actions. One key issue is that the process program represents a controlled walk through a design space and the run-time system as such maintains a representation of the

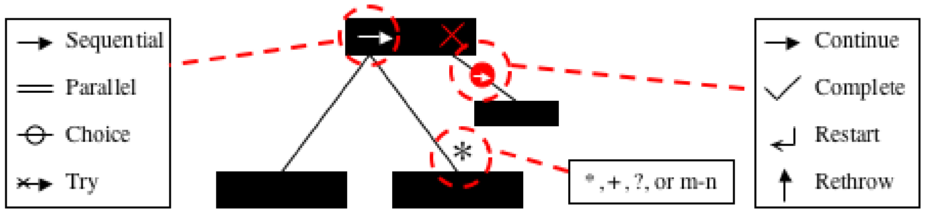

Fig. 1. Little-JIL Basics

state of this walk. Therefore, if we are to support undo, we need to do it in such a way that the run-time system can continue to maintain accurate process state information. Our approach is therefore to define undo semantics for the hierarchical process programs written in Little-JIL. This should give us robust semantics for cascading selective undo that can be extended to other dependency models.

In this section, we will briefly introduce Little-JIL, how Little-JIL programs represent control and data dependencies between tasks, and how we can use that information to support cascading selective undo.

2.1 Introduction to Little-JIL

Little-JIL programs are hierarchies of steps which are instantiated at run-time to become items to be placed on the agendas of agents acting in the process. The step kind, represented by the *sequencing badge* on the step in the graphical representation of the program, specifies how the instantiated sub-steps are to be sequenced. The badges for the four step kinds are shown in the left-most legend of Figure 1. Children of *sequential* steps are performed left-to-right with one child available for execution only after the previous one is completed. Children of *parallel* steps are performed in any, possibly overlapping, order. Only one child of a *choice* step is executed – when one of the steps is started at the discretion of the agents involved in the process, the other children are retracted. Children of *try* steps are attempted in left-to-right order and as soon as one of them *successfully* completes, the try step is completed. Program hierarchies are arbitrarily deep, with *leaf* steps representing the primitive actions available to agents.

Note that any undo mechanism for Little-JIL programs must allow a user to undo a primitive action that was *not* the most recently executed, for two reasons. First, Little-JIL programs can specify the coordination of actions of multiple agents and therefore multiple primitive actions can occur concurrently. At the minimum, an undo mechanism would have to allow an undo of one agent's most recent action (a *locally* linear undo) in cases where this is not the *globally* most recent action. Second, the completion of a primitive action in Little-JIL can cause new actions to become available. For example, if the action just completed is the last of the children of a sequential step, the sequential step is completed and possibly other actions are made available depending on the step kind of the parent of the sequential step. In order to maintain legal process state, undoing the user's most recent primitive action will require undoing the completion of the sequential step and the posting of the new options.

2.2 Modeling Control and Data Dependencies

The Little-JIL program structure shows control dependencies between steps – steps that share a sequential ancestor are sequentially ordered themselves. Data dependencies are modeled by parameter passing along the edges between parent and child steps. Steps have parameters specified as *in*, *out*, *in-out*, or *local*, and edges can be annotated to indicate where these parameters get their values. If an in or in-out parameter is bound from a parameter in the parent, its value is copied from the parent to the child before the child starts. Conversely, if an out or in-out parameter is bound to a parameter in the parent, its value is copied from the child to the parent when the child completes successfully. Therefore, to indicate a data dependency between two sibling steps, we bind the out parameter of one to a parent parameter, which is then bound to the in parameter of the other sibling.

Note that because data passing only occurs when steps start or complete, parameter passing can only be guaranteed between siblings if there also exists a control dependency between them. Siblings that are children of a parallel step, for example, can be started at the same time, so a parameter value copied from the parent might not reflect the value produced by a sibling. With this limitation, Little-JIL programmers indicate dependencies between tasks with control dependencies primarily. Note that in future versions of Little-JIL, a mechanism for sibling communication and synchronization will allow another way to specify task dependencies.

2.3 Cascading

Given a Little-JIL program from which a temporal sequence of primitive actions has been instantiated, and which defines the legal sequences of these same primitive actions, we can tell whether the resulting sequence of a selective undo would be meaningful according to the dependencies specified in the Little-JIL program. For example, if the leftmost child of the sequential step in Figure 1 were undone, sequential semantics would dictate the undoing of its siblings too. This can be done automatically and would avoid the problem with the "Car" class example given in Section 1 since the removal of the class would cause the removal of its methods as well.

3 Discussion

In this section we present important questions we plan to explore to determine undo semantics. First, is the Little-JIL language robust enough to capture all required semantics for undo? While we are currently working under the assumption that control dependencies subsume all important dependencies, data dependencies may exist where there are no statically-determined control dependencies. Also, control dependencies can be rather complex because of Little-JIL's additional semantics for cardinality and exception handling.

Cardinality specifies the number of instantiations to create for a step: zero or more (*), one or more (+), zero or one (?), or somewhere between m and n (m-n) for specified non-negative integers m and n. Exception handling specifies how to handle exceptions thrown from a child step, and then how to continue execution afterward. Once the exception is handled by executing a handler sub-step (which can have children), execution continues in one of four ways, as shown in the right-most legend of Figure 1: *continue* continues with the siblings of the sub-step that threw the exception, *complete* completes the step, *restart* restarts the step, causing the sub-steps to be re-instantiated, and *rethrow* causes the step to throw the exception to its parent.

With these rich semantics of control and data dependency, determining the semantics for cascading undo for arbitrary Little-JIL programs will be non-trivial.

A second question asks which kinds of actions should the user be allowed to undo. In [9], a distinction is made between undo as a regular user action (which itself can be undone), and a meta-command (which cannot). Allowing the user to undo an undo may prove difficult to represent to the user as an option, especially under the cascade model where multiple sub-tasks are involved. This also relates to the issue of how much of the cascade should be presented to the user when it occurs.

This question also pertains to the granularity of the task to be undone. Under the Little-JIL model, should selective undo be restricted to leaf steps or can internal nodes be undone? If the latter, this raises the issue of how much the user needs to know about Little-JIL semantics in order to effectively use the tool.

A third question relates to the mechanics of reversing an action when an undo is called for. Is reversability always decided at run time? Clearly this is sometimes necessary. For example, if an online purchase of airline tickets were to be modeled, undoing the purchase may or may not be allowed depending on the current date or type of ticket purchased. However, run-time evaluation reduces performance and for any domain, there will be undo steps that work identically in every instance and can therefore be statically embedded. Can a running little-JIL process with instantiated steps be augmented, perhaps with annotations, to reflect where dynamic evaluation of undo is necessary? In a related vein, could annotations be embedded into instantiated steps to provide more information about data dependencies (such as the time of purchase for the above example)? These annotations could provide information that would otherwise have to be inferred from the original step diagram or else not be available at all.

4 Conclusions and Future Work

In this paper we have begun to explore issues of selective undo in the context of sub-tasks which may have dependencies with one another. By using formal process modeling techniques, we have provided a means to capture both control and data dependencies. Our goal is to facilitate cascading selective undo in a conservative way, i.e. in a way that will always leave the user in a meaningful state while providing more flexibility.

It is important to note that context-dependent undo elicits other important issues besides the ones given in Section 3. For example, if a software designer renamed class X to Y, and then created a method "addYListener" and an interface "YListener" subsequently, then the undoing of the renaming step should cascade name changes in the method and interface too. The fact that these three constructs are related may originally only exist in the designer's mind and capturing that external knowledge is an important issue. However, *discovering* this knowledge is an AI problem that, while interesting, we do not wish to pursue at the onset. For now, we wish to focus on a conservative model of cascading where the knowledge required for a successful cascade are statically determined before moving on to more complicated problems of context dependency. In time, we wish to tackle these other issues as well.

References

1. Berlage, T.: A selective undo mechanism for graphical user interfaces based on command objects. ACM Transactions on Computer-Human Interaction **1** (1994) 269–294
2. Myers, B.A., Kosbie, D.S.: Reusable hierarchical command objects. In: Proc. of the ACM Conf. on Human Factors in Computing (CHI 96), ACM Press (1996) 260–267
3. Archer, Jr., J.E., Conway, R., Schneider, F.B.: User recovery and reversal in interactive systems. ACM Transactions on Programming Languages and Systems **6** (1984) 1–19
4. Abowd, G.D., Dix, A.J.: Giving undo attention. Interacting with Computers **4** (1992) 317–342
5. Mancini, R., Dix, A.J., Levialdi, S.: Dealing with undo. In: Proc. of INTERACT'97, Sydney, Australia, Chapman and Hall (1997)
6. Cass, A.G., Osterweil, L.J.: Process support to help novices design software faster and better. Technical Report 2005-018, U. of Massachusetts, Dept. of Comp. Sci. (2005)
7. Wise, A.: Little-JIL 1.0 Language Report. Technical Report 98-24, U. of Massachusetts, Dept. of Comp. Sci. (1998)
8. Wise, A., Cass, A.G., Lerner, B.S., McCall, E.K., Osterweil, L.J., Sutton, Jr., S.M.: Using Little-JIL to coordinate agents in software engineering. In: Proc. of the Automated Software Engineering Conf., Grenoble, France. (2000)
9. Vitter, J.S.: US&R: A new framework for redoing (extended abstract). In: SDE 1: Proc. of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, New York, NY, USA, ACM Press (1984) 168–176