EFFICIENT RECOVERY AND REVERSAL IN GRAPHICAL USER INTERFACES GENERATED BY THE HIGGENS SYSTEM

Scott E. Hudson
Roger King

Department of Computer Science
University of Colorado at Boulder
Boulder, CO 80309

Abstract

The Higgens user interface generation system being developed at the University of Colorado allows an interface designer to rapidly construct graphical user interfaces based on a primarily non-procedural interface specification. This paper discusses how user recovery and reversal, or Undo, is performed within a Higgens generated interface. A special data model is developed which has unique properties which combine to provide an efficient environment for implementing an undo mechanism. New algorithms based on recent work in incremental attribute evaluation are used to efficiently implement both the generated interfaces as a whole, and the undo mechanism in particular. In addition, a formal model of undo is used in an attempt to evaluate the power of the mechanism in order to compare it with other undo implementations.

Keywords: Recovery, Reverse Execution, Undo, Graphics Based Interfaces, User Interface Management.

1. Introduction

User reversal and recovery systems, sometimes called *Undo* systems, have recently been recognized as an important feature in user interfaces. Adding an Undo system to a user interface can have a profound effect on the useability and learnability of the interface. It allows the user to freely explore new or unfamiliar features of the interface without the normal fears of catastrophic mistakes. This greatly enhances the user's confidence, and allows more rapid learning. In addition, the capability to Undo actions allows the user to act in a more exploratory way. The user is able to try tentative actions which answer *what if* type questions without committing to those actions. This capability provides an entirely new level of functionality to a system, without changing any of its overt functionality.

Unfortunately, many previous Undo systems [Arch84, Vitt84] have been quite expensive. It has been necessary to produce a series of checkpoints which preserved part or all of some previous state of the system, along with a list of commands which are executed to move from one checkpoint's state to the next. Recovery was accomplished by restoring the state saved in some checkpoint, and reexecuting some saved commands, in order to return to the state requested by the user. Unfortunately, the process of creating a checkpoint is normally slow and often requires large amounts of space.

This work was supported in part by IBM under a Faculty Development Award (1984) and in part by Hewlett Packard under an American Electronics Association Faculty Development Program fellowship.

This paper discuses a technique for constructing efficient user recovery and reversal systems. This technique is employed by user interfaces constructed by the Higgens user interface generator. Higgens generated interfaces use a special data model to describe and implement the semantics of the interface, as well as the application itself. Interfaces constructed using this data model are described in a primarily non-procedural (rule-based) manner. Because of some special properties of this rule-based system, it is possible to construct simple inverses for all actions. This makes construction of a general undo system very easy and efficient.

In addition to being able to provide a powerful and general undo mechanism, the Higgens system has many other advantages for constructing graphical user interfaces. By using the algorithms described later (in section 3), it is able to automatically and efficiently construct graphical interfaces from a primarily non-procedural specification. In this way, the interface designer is free to concentrate on describing the behavior of the system without specifying precisely how or when the system will perform the computations necessary to update the graphical display. The system is able to use the rule-based interface specification to determine how to construct and update both the graphical images and the underlying application data, based on user actions.

In the next section we will talk about the Higgens interface generator and the goals behind it. Section 3 will discuss the powerful data model that underlies Higgens, and how it can be used to easily describe and implement powerful interfaces. Section 4 will discuss how undo can be implemented in a general yet very efficient manner within this data model. Section 5 will discuss the power and limitations of the undo mechanism provided, and finally, section 6 will discuss the current state of the implementation and provide conclusions.

2. Higgens

Higgens, the Human Interface Graphical Generation System, is a tool for automatically generating graphics based user interfaces. Higgens accepts a specification which is primarily non-procedural, and generates a user interface from it. Like some previous work done in generating human interfaces[Kasi82, Olse83] it borrows techniques from translator writing systems. Higgens, unlike most previous work in this area does not draw a sharp line between the application and the interface. It works with and has knowledge of the semantics of an application, as well of the interface itself.

One of the major goals of Higgens is to be able to support rapid and incremental development. At the present state of the art even if we use the best available design techniques, it is unlikely that we will be able to fully predict in advance all aspects of how real users will actually use a graphical interface.

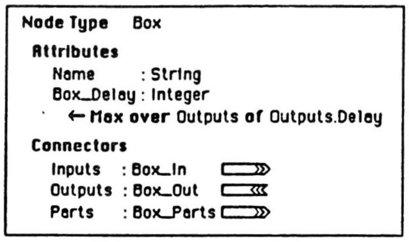
Consequently, it is not usually possible to create interfaces which have good human factors the first time. In order to produce high quality interfaces we are forced to test them with real users and change them to take the problems found into account. This means that we must often revise not only implementations, but also designs. This problem is particularly severe in the case of user recovery and reversal systems. This aspect of the implementation is traditionally a difficult one. If we are forced to reimplement the undo system each time the interface is modified, it will likely be too expensive to build.

Higgens overcomes this problem by allowing incremental development and by automatically providing an undo facility for all actions. Since interfaces are constructed as a set of semi-independent active entities, a partial working interface can be constructed and tested before the entire interface is finished. Only part of the semantics of each entity need be fully defined to construct a prototype interface which tests that part. In addition, since Higgens generated interfaces are described in a high level manner, changes are much easier to perform than they would be if conventional implementation techniques were used. As a result the designer is able to rapidly construct and test an interface in an incremental fashion, without having to discard all of the work that would have been put into one or more prototype systems. In addition, since Higgens provides a powerful undo facility as a primitive, the semantics of undo can be designed and implemented along with the items it affects, and does not need to be added later once the system is stable.

3. The Active Semantics Data Model

In order to understand how the Undo system works within a Higgens generated interface, it is necessary to understand the *active semantics data model* used to implement much of the functionality of the interface. In a Higgens generated interface, the semantics of both the application and the interface are described by this uniform data model. This data model encapsulates both the data of the application and its semantics. The model employs new algorithms adapted from techniques related to Knuth's attribute grammars[Knut68, Knut71] as well as from more recent work on incremental attribute evaluation[Deme81, Reps83] used in syntax directed editors. The state of the application and the interface are described in an *attributed graph*. Like the attributed trees often used in compiling, each node of an attributed graph has associated with it a number of *attributes* which describe the internal state of the semantic entity described by that node. In addition, *attribute evaluation rules* are given for computing certain attribute values as a function of other attributes within a given node, and from the values contained in related nodes. This allows the model to actively respond to its environment in a way which reflects its semantics. Finally, the interface designer may attach constraint predicates to attributes to perform error checking, and insure the integrity of the model. These constraints are automatically tested by the system and must always hold.

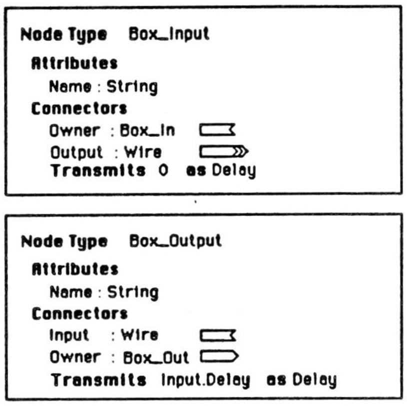
As an example of how we might use an attributed graph to model an application, we will examine a very simple gate level logic design system. In this system we will design a series of functional units called **Boxes**. As shown in figure 1, a **Box** consists of (is related to) a set of inputs, a set of output, and a set of parts (gates) which implement its function. For this example, we will be interested in finding the maximum delay time required for each box. Consequently, as shown in figure 1, we compute an attribute **Box_Delay**, which is defined as the maximum of the **Delay** values transmitted from the outputs.



```
Node Type   Box
 Attributes
   Name      : String
   Box_Delay : Integer
     ← Max over Outputs of Outputs.Delay
 Connectors
   Inputs  : Box_In
   Outputs : Box_Out
   Parts   : Box_Parts
```

Figure 1.

As shown in figure 2, these values are in turn derived from the values transmitted along the **Input Wire** relationship. These values come either from a **Box_Input** node (which always supplies 0), or from a **Gate** node. **Gate** nodes calculate a delay as a function of their operation, and logic family type, as shown in figure 3.

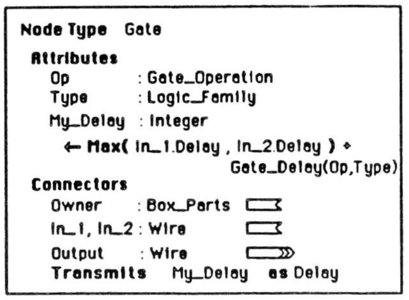


```
Node Type   Box_Input
 Attributes
   Name : String
 Connectors
   Owner  : Box_In
   Output : Wire
   Transmits  0  as Delay
```

```
Node Type   Box_Output
 Attributes
   Name : String
 Connectors
   Input  : Wire
   Owner  : Box_Out
   Transmits  Input.Delay  as Delay
```

Figure 2.

Because of the definition of the attributes in this graph, the model is able to respond automatically to changes. For example, if the user were to change the logic family type of a gate, the system would automatically determine which delay values were no longer correct, and recompute them. This will all happen without the implementor being forced to explicitly



```
Node Type   Gate
 Attributes
   Op       : Gate_Operation
   Type     : Logic_Family
   My_Delay : Integer
     ← Max( In_1.Delay , In_2.Delay ) +
                        Gate_Delay(Op,Type)
 Connectors
   Owner      : Box_Parts
   In_1, In_2 : Wire
   Output     : Wire
   Transmits   My_Delay   as Delay
```

Figure 3.

describe how or when the computations are to take place. Instead, the system can determine from the attribute evaluation rules exactly what attributes need to be recomputed when a change is made.

3.1. Levels of Interface

A Higgens generated interface is divided into 3 levels: application data, views, and abstract devices. The application data level encapsulates the data of the application and implements its semantics. The second level creates one or more *views* of the applications data. A view normally involves selection, filtration, and abstraction of information in order to specificly highlight or emphasize one aspect of the data. In addition, overall decisions about how data will be presented to the user, along with how the user may interact with the data, are done within views. Finally, the third level of abstract devices provides a very abstract and high level interface to the actual graphical I/O devices used.

Each of these levels is implemented using an attributed graph. This means that each level is able to respond to changes in other parts of the system in ways which are meaningful at that level. Entities at the abstract device level can translate the actions of physical devices and affect entities at the level of views. Actions on views can affect the way views are presented, and can be translated into actions on application data. Application data can then respond in ways that are meaningful to the semantics of the application they support. These responses can in turn affect views, abstract devices, and eventually the graphical images presented to the user. This process allows very powerful feedback to occur. This feedback can be not only on the normal lexical and syntactic levels, but can also be on a deeper level which reflects the semantics of the underlying problem domain. It is this powerful semantic feedback which guides the user into forming the helpful mental models needed for good interfaces. In addition, since the specification of the attribute evaluation rules which control this feedback is primarily non-procedural, the designer is free to concentrate on what the feedback will be, and may leave many of the details of how and when it is carried out to the system.

3.2. Translating Data Into Views and Images

In order to construct graphical images from data graphs, the conventional approach would be to use programs which traversed the graph extracting information, making decisions, and producing partial images as the traversal proceeds. Such a traversal can in general be very powerful and flexible, since the nature of the traversal can be determined in arbitrarily complex ways based on the actual data encountered in the graph. Higgens uses a traversal process similar to this conventional approach in order to translate its attributed data graphs into views.

A Higgens interface specification contains a series of *traversal plans*. These plans determine how traversals proceed based on predicates over the attributes of nodes they visit. However, unlike conventional traversals which exist only as a dynamic series of procedure invocations, Higgens traversals are explicitly represented as data objects. Each visit of a data node is represented by a *viewing* node (thus forming a tree of viewing nodes). These nodes are normal data objects. They are given attributes and attribute evaluation rules. They are persistent, and have access to the attributes of the data node they visit. In this way, they are able to implement the view dependent semantics needed to provide selection, filtering, and abstraction of the underlying data. In addition, each node may render images and accept input from the abstract devices used to implement the graphics of the interface.

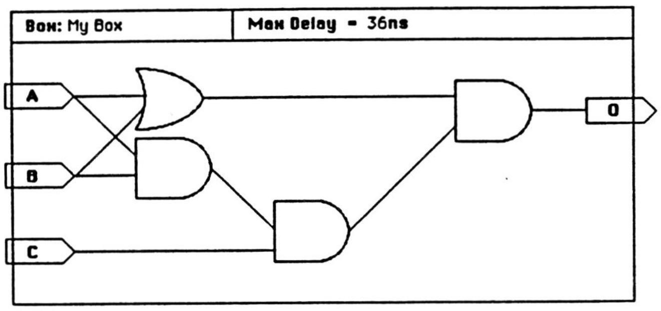


Figure 4.

For example, in our simple logic design system, the user might want one of several different views of a box. These views could range from a simple summary view showing just a rectangle with the box's name in it, to a complete view like the one in figure 4, showing all inputs, output, and gates which implement the box. Each of these views can be constructed by a different traversal.

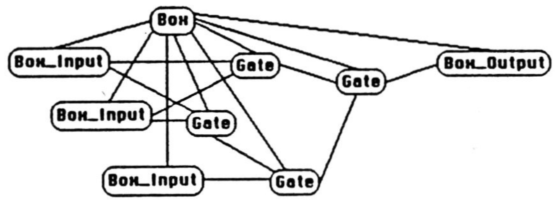


Figure 5a.

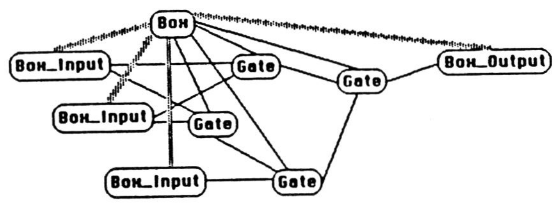


Figure 5b.

As an intermediate example, we will consider a view which shows just the name, maximum delay, inputs, and outputs of a box without showing the gates used to implement it. Figure 5a shows an example data graph corresponding to the view given in figure 4. Figure 5b shows the path that a traversal would take in order to implement our sample view. This path visits the **box** node itself, then each of the inputs, then finally the single output node. Each of these node visits is represented by a viewing node. Figure 6 shows several of these viewing nodes as they would be created by a traversal.

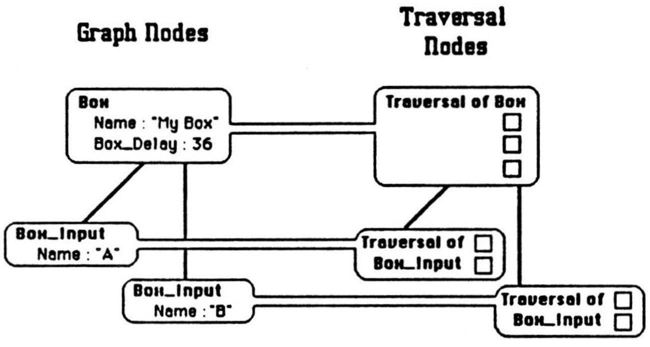


Figure 6.

In order to deal with actual graphical I/O devices, a viewing node communicates with a series of abstract devices. These abstract devices are constructed using the *Planit* picture planning language which is a part of Higgens. Planit allows the interface designer to construct *picture plans*. A picture plan consists of a specification of how (hierarchical) images are to be drawn, and how logical input devices are to be constructed. These specifications describe images and devices based on a set of formal parameters. A viewing node gives a picture plan and provides a set of actual parameters in order to instantiate an abstract device. A viewing node may then at any time change these actual parameters. The images and devices controlled by the picture plan will then be updated to reflect these new actual parameter values. The parameters provided by a viewing node may be intrinsic or derived attributes of the node. In this way, when they are changed, or recomputed, they will directly affect the picture presented to the user. Since picture plans may contain conditionals based on arbitrary expressions of parameters, as well as calls to other picture plans, they can be given arbitrarily complex behavior. Figure 7 illustrates how picture plans would be attached to the viewing nodes of our example.

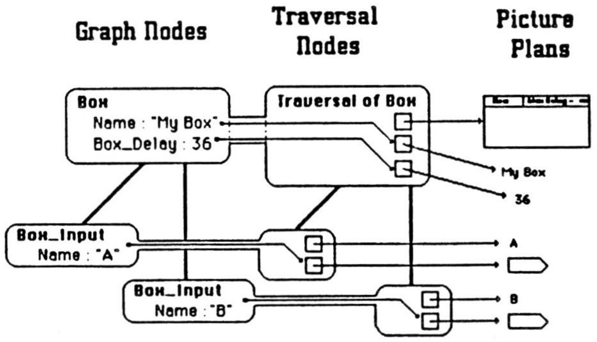


Figure 7.

In addition to presenting output images to the user, abstract devices can respond to user inputs by means of messages sent to viewing nodes. Viewing nodes respond to these messages by invoking editing commands. These editing commands can modify attributes, create and delete data nodes, as well as establish and break relationships within the data graph. In this way, views may translate the actions of devices into actions appropriate to the semantics of the underlying data.

In a Higgens generated interface, feedback can occur at all levels. The user can use actual graphical I/O devices to manipulate the abstract devices presented. The abstract devices send messages to views, which in turn may modify their own attributes, and those of applications nodes in order to carry out user requested actions. These modifications in turn invoke whatever computations are needed in order to satisfy the attribute evaluation rules of the system. These computations propagate their effects throughout the system, and may result in changes to views, or to the traversals that construct views. This in turn may change the parameters of the abstract devices they control. This can finally change the images and logical devices presented to the user. Thus, it is possible for feedback to occur locally at each level, as well as across several levels. In this way, feedback can go beyond the properties of the graphical entities used to present the data, and is able to convey the semantics of the underlying data.

3.3. The Attribute Evaluation Algorithm

Whenever changes are made to some part of the attributed graph, the system must ensure that all attributes retain a value which is consistent with the attribute rules given by the interface designer. This requires some sort of an attribute evaluation algorithm. One approach would be to recompute all attribute values every time a change is made to any part of the system. This is clearly too expensive. What is needed is an algorithm for incremental attribute evaluation, which computes only those attributes whose values change as a result of a given modification. This problem also arises in the area of syntax directed editing systems, so it is not surprising that algorithms exist to solve this problem for the attribute grammars used in that application. The most successful of these algorithms is due to Reps. [Reps82] Reps' algorithm is optimal in the sense that only attributes whose values actually change are recomputed, and that the total overhead of the algorithm is $O(|Changed|)$, where Changed is the set of attributes whose values actually change.

Unfortunately, Reps' algorithm, while optimal for attributed trees, does not seem to extend directly to the arbitrary graphs used by a Higgens generated interface. Instead, a new incremental attribute evaluation algorithm has been designed for use with Higgens. This new algorithm is simpler, and exhibits similar behavior to Reps' algorithm. In particular, for any given change it will never recompute an attribute that would not have been recomputed by Reps' optimal algorithm. However, it does have a slightly inferior worst case upper bound on the amount of overhead incurred.

The algorithm works by using a strategy which first determines what work has to be done, then performs the actual computations. The algorithm uses the dependencies between attributes. An attribute is dependent on another attribute if that attribute is mentioned in its attribute evaluation rule (i.e. is needed to compute the derived value of that attribute). When the value of an intrinsic attribute is changed, it may cause the attributes which depend on it to become *out of date* with respect to their defining attribute evaluation rules. Instead of immediately recomputing these values, we simply mark them as out of date. We then find all attributes which are dependent on these newly out of date attributes, and mark them out of date as well.

This process continues until we have marked all affected attributes. During this process of marking, we determine if each marked attribute is *important*. Attributes are said to be important if they have a constraint predicate attached to them, or if part of the current graphical display depends on them. When we have completed marking attributes during the first phase of the algorithm, we will have obtained a list of attributes which are both out of date and important. We can then use a demand driven algorithm to evaluate these attributes in a simple recursive manner. The calculation of attribute values which are not important may be deferred, as they have no immediate effect on the interface or the application. If user actions cause abstract devices or views to be changed, new attributes may become important, and new computations of out of data attributes may be invoked in order to obtain the values needed to construct new displays.

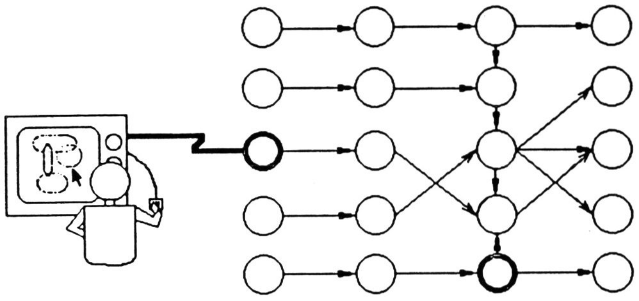


Figure 8.

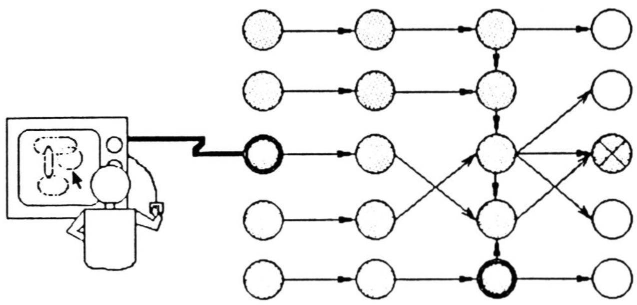


Figure 9.

In figure 8, we have presented a set of attributes in order to illustrate how the attribute evaluation algorithm works. The circles represent individual attributes. How these attributes are distributed among nodes is not important to the evaluation algorithm, only the dependencies between attributes are important. In this case, two attributes have been designated important. One provides a parameter to a picture plan which controls part of the current display. The other has a constraint predicate attached. For this example, we will presume that the attribute marked with an X in figure 9 has been changed by some user action. The first phase of the algorithm responds to this change by marking a series of attributes as out of date. These attributes have been marked in gray in figure 9. Notice that along with a number of other nodes, the two important nodes have been marked out of date. These nodes will be remembered for use in the second phase of the algorithm. During the second phase, the system will attempt to obtain correct values for all important attributes using a simple recursive evaluation strategy. The evaluation starts with each important attribute which is also marked out of date, and recursively evaluates only those attributes need to obtain the original important attribute value. The attributes that are reevaluated in the second phase for our example are marked in figure 10. Once the second phase of the algorithm is complete, the graph will be left in the state shown in figure 11. Notice that a number of attributes are left with out of date values. The values of these attributes cannot affect the observable state of the system, therefore, we can safely defer their computation. If the same change made here is done several times, the system will not be forced to recompute these values several times despite the fact that none of them will actually be used. Instead, the attributes will retain their old out of date values until the correct values are actually needed.

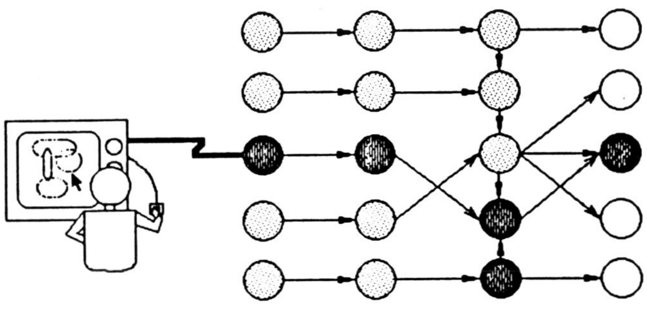


Figure 10.

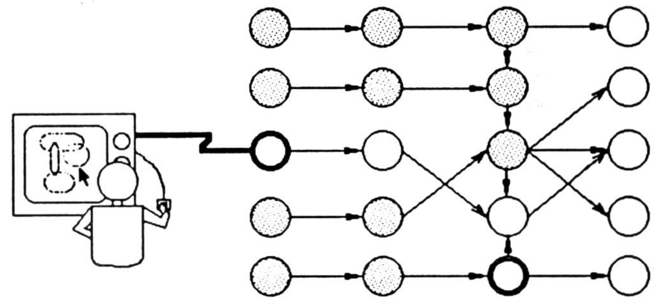


Figure 11.

In order to support actions which change not only the attributes of the application data but also, its structure (i.e. the relationships between entities) a process similar to that used for intrinsic attribute changes is used. When a relationship is broken, the system determines which derived attributes depend on values that are passed across the relationship. These attributes are marked out of date just as if an intrinsic attribute had changed. When a relationship is established, the second half of the attribute evaluation algorithm is invoked to evaluate attributes which are out of date and important. In order to insure that derived attributes can always be given a valid value, and hence a display generated, the system insures that relationships are not left dangling across attribute evaluations. This is either done explicitly by application supplied actions, or where necessary the system will provide special dummy nodes to tie off any dangling relationships.

During the evaluation of attributes, certain attributes will have constraint predicates attached to them. By attaching a constraint predicate to an attribute, the interface designer is able check for error conditions, and insure the semantic integrity of all entities. After an attribute is evaluated, any attached constraint predicates are tested. If any of these evaluates false, a constraint violation exists. By default, this causes the user command invoking the evaluation to fail and be undone. Optionally, a special recovery action associated with the constraint can be invoked to attempt to recover from the violation. In either case, the constraint must be satisfied or the user command invoking the evaluation will fail and be undone. The next section will discuss how user recovery and reversal is actually performed, and how the boundaries between user commands are established.

4. Execution and Reversal

It would seem that the large amounts of derived data, along with the fact that the order of computations is not defined, and that some computations are deferred, would make creating a general user reversal and recovery system a difficult task. However, as it turns out, the properties of the active semantics data model combine to provide an excellent environment for efficient implementation of a user recovery and reversal system. Note that the set of attribute evaluation rules can be used at any time to obtain the values of derived attributes from the set of intrinsic attribute values. This means that in order to Undo commands which simply change attributes, the only action needed is to restore the old values of those attributes changed, and invoke the normal attribute evaluation mechanism used to respond to these changes. The system will restore itself to its old state automatically. Similarly, commands which change the structure of the attributed graph, need only remember the old structure in order to allow for their complete reversal. It is this simplicity which makes user reversal and recovery efficient in a Higgens generated interface.

4.1. Primitive Commands

We can now examine the kinds of primitive commands needed to effectively deal with the attributed graphs used to implement the interface, and how the effects of each of these primitives can be reversed. Clearly we need a command to replace the value of an intrinsic attribute. We also need a command to create a new node of a given type, to delete a node, to break a connection between nodes, and finally to establish a new connection between nodes. We will call these primitive commands **assignment**, **create**, **delete**, **cut**, and **connect**. Notice that each of these primitive commands has a simple inverse as shown below.

Operation	Inverse
Assignment	an **assignment** which restores the old value of the attribute
Create	a **delete** of the **created** node
Delete	replacement of the deleted node
Cut	a **connect** between the affected nodes
Connect	a **cut** of the affected relationship

Because each primitive command has a simple inverse, we are not forced to determine or remember the full effects of each command to be able to reverse its action. Instead, we need only remember its inverse. The system is able to derive the reversal of its full effects in the same way it derived the effects to begin with.

In addition, we can use this same capability to provide a Redo mechanism. The user may wish to undo an undo; that is they may wish to redo the original operation. Since all primitive operations have an inverse which is also a primitive command, the system can reconstruct original commands from their inverses in much the same way that the inverses were constructed to begin with. In this way, the system has the ability to redo commands after they have been undone. Unfortunately, whenever new commands are executed after an undo, the system will be unable to redo old commands, since the new commands may have destroyed something which the old commands depend upon. As we will see in section 5, this places a limit on the power of the undo mechanism provided.

In order to make effective use of primitive commands to act upon attributed graphs, we need to embed them in control structures. We need the conditionals, loops, and procedural abstraction mechanisms found in more traditional programming languages. However, in order to reverse commands, we need only record the sequence of inverses for the primitive commands actually executed, without regard for the control structures which arranged for their original execution. Using these inverse commands, it is possible to fully reverse the effect of commands without explicitly reexecuting their control structures.

4.2. User Level Commands

Although the primitive commands described above provide a good mechanism for the designer to describe the implementation of the interface, they are not suitable for use directly as user commands. Instead, user commands are normally built as a series of primitive commands embedded in control structures. From the user's point of view, however, this internal structure is invisible, and commands appear as atomic units.

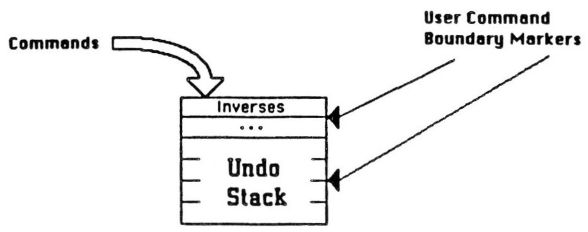


Figure 12.

Because of this, it would be highly inappropriate to undo and redo individual primitive commands. Instead commands should be undone and redone in groups which correspond to what the user considers atomic operations. In order to accomplish this, Higgens provides a means for declaring the boundaries of user commands. As shown in figure 12, as operations are performed, their inverses are automatically saved on an undo stack for possible later reversal. In order to allow the manipulation of whole user commands instead of primitives, Higgens allows the interface implementor to use a special command which marks the boundaries of user commands on the undo stack. Later when an undo action is invoked, Higgens will undo all primitive commands up to the last mark and construct a corresponding redo stack for those commands. As shown in figure 13, an undo operation performs the saved inverse operations, and pushes the recreated original commands onto the redo stack. Similarly, if a later redo action is performed, Higgens redoes all primitive commands up to the last command mark, and places their inverses on the undo stack. In this way, the interface designer is able to implement user commands using whatever primitives and control structures are needed, while still allowing the system to undo actions in chunks which are meaningful to the user.

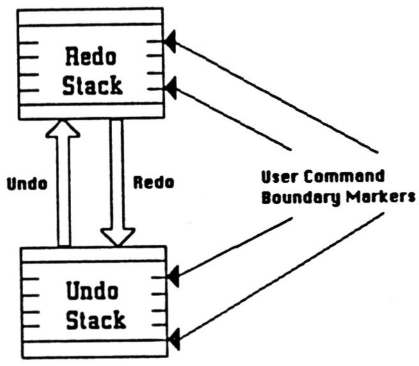


Figure 13.

5. Power of the Higgens Undo Mechanism

In[Gord84] an elegant formal model of undo systems is constructed which allows us to compare and characterize the power of various undo strategies. Their model makes use of the following formal entities:

S	a set of extended states (including history information)
c(s)	the contents of an extended state s (excluding history information)
O	a set of operations mapping extended states to extended states [denote applying operation k to state s as ks]
u in O	an Undo operation
K	a subset of O not including u

Notice that a state s in S includes whatever history of previous commands is needed in order to be able to perform undo operations. In a Higgens generated system this includes the attributed graph as well as the undo and redo stacks (which we will refer to as the *history*). The expression c(s) is used to denote the state s without its associated history. In a Higgens generated system this includes just the attributed graph.

Based on these entities, we can define a formal property which captures our intuitive notion of undo. This property is the Basic Undo Property.

Basic Undo Property:
$c(uks) = c(s)$ for all s in S and all k in K

In addition to this basic property, Gordon et.al. define two other important properties that we would like undo systems to have:

Thoroughness:
$c(o_n ... o_1 uks) = c(o_n ... o_1 s)$
for all s in S and o_i in O (possibly including u)

Invertibility:
$c(rus) = c(s)$ for all s in S and some r in O

Intuitively, the thoroughness property insures that an undo is complete, that the state returned to by using an undo can in no way be distinguished from the original state (even when we consider the behavior of the undo or redo operators). Invertibility insures that a redo operation (r) exists, so that undo operations are not themselves irreversible. Both these properties are highly desirable, unfortunately, it is can be show that no non-trivial system can be both thorough and invertible. As a consequence Gordon et.al. explore a somewhat weaker form of thoroughness called unstacking.

Unstacking:
$c(u^n f_n ... f_1 s) = c(s)$ for f_i in K and any positive n

It can be shown that systems with separate undo and redo operations can be both unstacking and invertible. In fact, these are precisely the properties that a Higgens generated interface has. The user can undo arbitrary sequences of operations by repeated applications of the undo operation, and can redo operations so long as new operations are not performed. More formally, the Basic Undo Property holds since

$c(upg) = c(g)$ for any primitive operation p and any attributed graph (and history) g

The Invertibility property holds since Higgens supports a redo operator r such that

$c(rug) = c(g)$ for any attributed graph and history g

The Unstacking property holds since

$c(u^n p_n ... p_1 g) = c(g)$ for any primitive operations p_i

but, the Thoroughness property fails since

$c(rrupupg) <> c(ppg)$

It is not clear at this point if unstacking is the most powerful property that might be achieved in conjunction with invertibility (while still retaining the basic undo property). However, it does seem clear that it is at least near what we can hope to achieve given the theoretical limits that exist, and it compares favorably with existing undo systems.

6. Conclusion

The Higgens system is currently being implemented at the University of Colorado at Boulder. A prototype of the abstract device description language Planit which makes up part of Higgens, has been completed. The current implementation runs on a Silicon Graphics IRIS display device connected to a VAX, and on a SUN workstation. The remainder of the system is under development and is scheduled for completion in middle to late 1985.

In this paper we have discussed the user recovery and reversal mechanism used in the Higgens interface generation system. Despite the very powerful data model used to implement Higgens generated interfaces, they provide an efficient undo mechanism in a way which is fully integrated with the rest of the system. This technique greatly improves the useability of human interfaces, and makes learning to use these interfaces much easier.

Acknowledgements

The authors would like to thank Dan Olsen of BYU. The original form of the incremental attribute evaluation algorithm we use in the Higgens system emerged from a long discussion between Dan and one of the authors. The authors would also like to thank Clayton Lewis for introducing us to the formal notion of undo that he and his colleagues have developed.

References

Arch84.
Archer, James E. Jr., Richard Conway, and Fred B. Schneider, "User Recovery and Reversal in Interactive Systems," *ACM TOPLAS* **6** pp. 1-19 (Jan. 1984).

Deme81.
Demers, Alan, Thomas Reps, and Tim Teitelbaum, "Incremental Evaluation for Attribute Grammars with Application to Syntax Directed Editors," *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages,* pp. 105-116 (Jan. 1981).

Gord84.
Gordon, Robert F., George B. Leeman, and Clayton H. Lewis, "Concepts and Implications of Interactive Recovery," *IBM Tech Report RC 10562 (#47293),* IBM Thomas J. Watson Research Center, (June 1984).

Kasi82.
Kasik, D. J., "A User Interface Management System," *Computer Graphics* **16** pp. 99-106 (July 1982).

Knut68.
Knuth, D. E., "Semantics of Context-Free Languages," *Math. Systems Theory J.* **2** pp. 127-145 (June 1968).

Knut71.
Knuth, D. E., "Semantics of Context-Free Languages: Correction," *Math. Systems Theory J.* **5** pp. 95-96 (Mar. 1971).

Olse83.
Olsen, Dan R. and Elizabeth P. Dempsey, "SYNGRAPH: A Graphical User Interface Generator," *SIGGRAPH '83 Conference Proceedings,* pp. 43-50 (July, 1983).

Reps82.
Reps, Thomas, "Optimal-time Incremental Semantic Analysis for Syntax-directed Editors," *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages,* pp. 169-176 (Jan. 1982).

Reps83.
Reps, Thomas, Tim Teitelbaum, and Alan Demers, "Incremental Context-Dependent Analysis for Language-Based Editors," *ACM TOPLAS* **5** pp. 449-477 (July 1983).

Vitt84.
Vitter, Jeffrey, "US&R: A New Framework for Redoing," *SIGPLAN Notices* **19** pp. 168-176 (May 1984).