TEXT EDITING AND BEYOND: A STUDY IN LOGIC MODELING

by

Michael Bieber
Computer Science Department
Carroll School of Management
Boston College
Chestnut Hill, Massachusetts 02167-3808

and

Tomás Isakowitz
Information Systems Department
Leonard N. Stern School of Business
New York University
New York, New York 10003

December 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-91-39

forthcoming in *Decision Support Systems*

Abstract

This paper presents a logic modeling exercise in which we develop, test and implement a logic model for a text editor and use it to test existing text editing software. We begin by presenting a first order Horn logic axiomatization of a text editor by providing domain equations for the primitive operations *insert*, *delete* and *character retrieval*. We show that this logic model captures the essential aspects of the text editing task and how more complex features are built using these primitives. We discuss possible implementations and conclude that any operational semantics—the set of algorithms that perform the task—must be strongly related to the logic model we present. In other words, each operational semantics constitutes a *model* of the logic theory. Next, we illustrate the usefulness of the model by implementing a basic text editing system and testing the correctness of an existing text editor. We conclude by describing how we are integrating these modeling techniques into a larger and more complex knowledge-based system.

Keywords:

Logic Modeling, Text Editing, Operational Semantics, Hoare Logic, Horn Logic, Prolog, Hypertext

Michael Bieber is an Assistant Professor of Information Systems at Boston College, Carroll School of Management. He received his B.S.E. in Computer Science and Ph. D. in Decision Sciences at the University of Pennsylvania. His research interests include hypertext, information presentation and knowledge-based decision support.

Tomás Isakowitz is an Assistant Professor at the Information Systems Department of the Leonard Stern School of Business, New York University. He received his B.Sc. in Mathematics at the Hebrew University of Jerusalem, his M.Sc. in Mathematics at the University of California at Santa Barbara, his M.S.E. and his Ph.D. in Computer Science at the The Moore School of Electrical Engineering, University of Pennsylvania. His research interests include logic programming, decision support systems and hypertext systems.

1 Introduction

Computer applications can be very complex and need to be planned well. Logic modeling [1] can be a useful tool in the design and testing of a computer system. A logical axiomatization of the various components of a system can provide several useful services:

- Logic can serve as a language for expressing the mathematical semantics, assumptions and concepts of a system's domain.
- As a design tool, the development of a formal logic model forces one to determine and to explicitly express all of the elements in the domain of the system.
- A logic model can be used to build a prototype system in a programming language such as Prolog [2,3] or OBJ [4]. This prototype will prove useful in testing the ideas behind the specification of the logic model.
- Axioms in the logic model can be used to test the correctness of specific implementations.
- Production rules—one of the major knowledge representation schemes used in artificial intelligence and expert systems [5]—can be represented directly in first order logic.

In this paper we present a logic model for a text editor. Text editing is a necessary feature in any computer system which accepts textual input from the user. Our logic model of a text editor can be implemented directly as a standalone computer application (which we have done in section 4). Alternatively it could be used as a basic text-handling module, or as the foundation of a more comprehensive module in a larger computer system model (as we show in section 6).

In section 2 we develop a first order logic Horn axiomatization of a text editor. We show how all text editing commands are built upon the insert, delete and character retrieval operations. Section 3 describes an operational semantics for representing the components of a text editing system. We show that this operational semantics is sound and complete with respect to the logic model. In section 4 we use the axioms and operational semantics to implement a text editor of our own in Prolog. In section 5 Hoare logic is used to test the correctness of an existing text editor based upon the axioms developed in section 2 and implemented in section 4. We conclude with a general discussion of logic modeling for constructing large information systems and illustrate this with an example of how we have expanded this model.

We begin by presenting two logic models for text editing, the first quite simple, the other more flexible.

2 A Logic Model for Text Editing

In this section we provide two theories in predicate calculus that model the principal operations a text editor has to perform. Our language is first order logic with equality. We assume that the reader is familiar with the basic concepts in logic ([6,7]). There is a set of variables that, following the Prolog convention, we shall denote by uppercase letters: $X, Y, Z, \ldots$, a set of constants denoted by lowercase letters from the beginning of the alphabet: $a, b, \ldots$, and function symbols also denoted by lowercase letters: $f, g, h, \ldots$ The logical connectives are $\wedge$ for "and", $\vee$ for "or", $\neg$ for negation, $\supset$ for implication, $\equiv$ for equivalence, $\forall$ for universal quantification over variables and $\exists$ for existential quantification. Terms and formulae are defined over this language. There is a special predicate $\doteq$ which stands for equality and always is to be interpreted as such. We write $\doteq$ instead of the normal equality symbol to emphasize the fact that it is part of the logical language.

In section 2.1 we present a "bare bones" text editor in the form of a stack. Its purpose is to get our feet wet, and to introduce logic modeling and some of the constructs upon which we shall be expanding. In section 2.2 we increase the efficiency of the bare bones editor by allowing direct access to individual characters instead of being restricted to the head of a stack. Each of the more advanced editor's commands are axioms built from a combination of three basic functions: *insert*, *delete* and *retrieve character*, and we show this construction. To be able to implement such a model in logic programming languages such as Prolog we must show that this model can be reduced to "Horn logic". We discuss this further in section 2.3.

2.1 A Bare Bones Text Editor

We begin this study with a set of operators corresponding to text editing functions and by capturing their behavior in a logical way. This logical way is given via axioms in the form of equations. As an example, consider the editor that interacts with the user when he or she is typing a command to the operating system of a PC (e.g., MS-DOS). In general, this editor allows the user to enter characters one by one and to backspace. The only means of correcting something is by backspacing. In summary, it supports addition and deletion of characters only at the end of the file. We represent this simple model with two functional operators:

- $\mathrm{a}(F, C)$ which returns the file resulting from appending character C to the end of file F, and
- $\mathrm{b}(F)$ which returns the file resulting from backspacing over the last character of file F.

One necessary axiom specifies that an append followed by a backspace leaves the file unchanged:

$$\forall C \forall F\ \mathrm{b}(\mathrm{a}(F, C)) \doteq F.$$

In general all variables appearing in an axiom will be universally quantified and from now on we shall not write the quantification explicitly. Next we specify the result of backspacing past the beginning of the file. Using the constant λ to denote the empty file, we have a second axiom:

$$\mathrm{b}(\lambda) \doteq \lambda.$$

For example, the sequence of operations corresponding to writing the characters i, t, i and s followed by two backspaces is represented by the term:

$$\mathrm{b}(\mathrm{b}(\mathrm{a}(\mathrm{a}(\mathrm{a}(\mathrm{a}(\lambda, i), t), i), s)))$$

which can be proven to be equivalent to the term: "$\mathrm{a}(\mathrm{a}(\lambda, i), t)$" by two applications of the first axiom. This last term represents the file containing the characters: "*it*."

2.2 A More Advanced Editor

We now expand these concepts to address the issue of arbitrary insertions and deletions at any position of the file. We represent character positions with non-negative integers. We assume the theory of natural numbers is available in the form of some axiomatization which includes the predecessor, successor and the natural ordering "<" functions (e.g., Presburger's Arithmetic). We abbreviate $pred(P)$ by $P - 1$ and $succ(P)$ by $P + 1$. The function symbols are:

- ins(C, F, P), which represents the file resulting from *inserting* character C into file F at position P (i.e., before the character currently residing at position P),
- del(F, P), which represents the file resulting from *deleting* the character at position P in file F, and
- ch(F, P), which represents the character *retrieval* at position P in file F.

Many other operations can be built on top of these basic ones. For example, deleting a block of characters from position p_1 through p_2 is equivalent to successively deleting the characters at positions $p_2, p_2 - 1, p_2 - 2, \ldots, p_1$. We can define this recursively with the axioms:

$$\begin{aligned} \text{del-block}(F, P, P) &\doteq \text{del}(F, P) && (1) \\ \text{del-block}(F, P_1, P_2) &\doteq \text{del-block}(\text{del}(F, P_2), P_1, P_2 - 1) \\ &\quad\;\; \text{if } P_2 > P_1. && (2) \end{aligned}$$

Equation 2 is a *conditional equation* of the form "$t \doteq t'$ if C", where C is a formula. The way these axioms are used in deductions is by first making sure the condition holds and only then applying the axiom. Suppose that our theory (the set of axioms) Γ contains the conditional equation "$t \doteq t'$ if C." If C holds in Γ (i.e., $\Gamma \vdash C$) we can deduce that $t \doteq t'$ holds in Γ (i.e., $\Gamma \vdash (t \doteq t')$).

Similarly one can define a copy operation from one file to another by successively performing "ch" and "ins" operations, as in "ins(ch$(F_1, P), F_2, P'$)."

The operation of cutting can be represented by first copying a block to a special file, e.g., the *clipboard*, and then deleting the block. Pasting equates to copying from the special file into the current one. Similarly one can define search, replace and all other text editing operations. (We are excluding "input/output" procedures such as saving the file, beeping to the user, printing on screen, window operations and monitoring for keyboard activity because these are not intrinsically related to the text editing process from the modeling point of view.)

We are not suggesting that this is the best way to implement text editing operations, but rather that their logical nature is captured by such axioms. The point is that, however these operations are implemented (that is, whatever the operational semantics are), these axioms have to be satisfied. In section 5 we show how they can be used to ensure correctness of a specific implementation of a text editor.

Now, it is not enough to say that "del" stands for delete and "ins" for insert, we have to impose some structure to enforce this interpretation. We do this with the following set of axioms that establish the relationships among the different operators. In defining these relationships we shall use two constants, which were not in our original language specification: λ and *eof-char*, denoting the *empty file* and an *end of file* marker respectively. We also assume a lexicographic ordering on characters where the *eof-char* character is the smallest. (This permits us to avoid using negation which, as we shall see, would have made our formulation non-Horn.) For convenience we also introduce an *end of file* function "eof(F,P)," which returns the boolean *true* if integer position P is past the end of the file of F, i.e., beyond the actual *eof-char*, which is the final character in a file. We define this in axiom 3 below. Note that the *ch* function returns *eof-char* for any position past the end of file. (This can be proven using axioms 4 and 10.)

$$\begin{aligned} \mathrm{eof}(F,P) \equiv P > 0 \wedge (ch(F,P-1) &\doteq \text{eof-char}) && (3)\\ \mathrm{ch}(\lambda,P) &\doteq \text{eof-char} && (4)\\ \mathrm{ins}(C,F,P) &\doteq F \quad \text{if } \mathrm{eof}(F,P) && (5)\\ \mathrm{del}(F,P) &\doteq F \quad \text{if } \mathrm{eof}(F,P+1) && (6) \end{aligned}$$

Equation 4 states that character retrieval from any position of the empty file produces an end of file marker. Equation 5 tells us that inserting beyond the end of file has no effect and returns the original file F. Equation 6 ensures that we cannot delete the end of file marker or at any position after it. Again, this just returns the original file.

With these defined we can declare the relationships among the different functions. First we define the relationship between the operations of inserting and retrieving characters.

$$\begin{aligned}
\mathrm{ch}(\mathrm{ins}(C,F,P),P) &\doteq \text{eof-char} \\
&\quad \text{if } \mathrm{eof}(F,P) && (7)\\
\mathrm{ch}(\mathrm{ins}(C,F,0),0) &\doteq C && (8)\\
\mathrm{ch}(\mathrm{ins}(C,F,P),P) &\doteq C \\
&\quad \text{if } P > 0 \wedge \\
&\quad \mathrm{ch}(F,P-1) > \text{eof-char} && (9)\\
\mathrm{ch}(\mathrm{ins}(C,F,P),N) &\doteq \mathrm{ch}(F,N-1) \\
&\quad \text{if } P < N && (10)\\
\mathrm{ch}(\mathrm{ins}(C,F,P),N) &\doteq \mathrm{ch}(F,N) \\
&\quad \text{if } P > N && (11)
\end{aligned}$$

Equations 8 and 9 state that the character retrieved at position P is the same as that which was inserted there. Equation 7, which follows from equations 3, 4 and 10, makes an exception for the case in which the insertion was illegal (i.e., beyond the end of file). Equations 10 and 11 describe the movement of the original characters positioned before and after an insertion point. Equation 10 states that characters at positions N after an insertion point P are shifted over by one position (now they are at position N whereas previously they were at position $N-1$). Equation 11 states that characters located in front of an insertion point remain unchanged by an insertion.

Next we declare the relationship between the insert and delete functions. If we always start with an empty file we never have net deletions. This is ensured by axiom 6 and by the next group of axioms, which enables us to cancel deletions from terms by finding corresponding insertions and removing matching occurrences.

$$\mathrm{del}(\mathrm{ins}(C,F,N),N) \doteq F \qquad (12)$$

$$\begin{aligned} \text{del}(\text{ins}(C, F, P), N) &\doteq \text{ins}(C, \text{del}(F, N-1), P) \\ &\quad \text{if } P < N && (13) \\ \text{del}(\text{ins}(C, F, P), N) &\doteq \text{ins}(C, \text{del}(F, N), P-1) \\ &\quad \text{if } P > N && (14) \end{aligned}$$

As an example, start with an empty file, add the string ab at positions 0 and 1, and insert c at position 1 between them. This yields the file acb and can be represented by the term:

$$t \equiv \text{ins}(c, \text{ins}(b, \text{ins}(a, \lambda, 0), 1), 1).$$

If we now delete the character at position 0 the result should be the file cb. The deletion is represented by the term:

$$t' \equiv \text{del}(\text{ins}(c, \text{ins}(b, \text{ins}(a, \lambda, 0), 1), 1), 0).$$

By axiom 14 we transform this term to:

$$\text{ins}(c, \text{del}(\text{ins}(b, \text{ins}(a, \lambda, 0), 1), 0), 0).$$

Applying the same axiom once more we obtain:

$$\text{ins}(c, \text{ins}(b, \text{del}(\text{ins}(a, \lambda, 0), 0), 0), 0)$$

which is equal to "$\text{ins}(c, \text{ins}(b, \lambda, 0), 0)$" via axiom 12, producing the file cb.

Certain properties hold in the axioms above. For example, deleting a character and then reasserting it results in an unchanged file. This is something natural that we expect to hold and does follow from the above axioms. Equations 7 through 11 could be compressed into a single equation using an *if... then... else* construct, as could equations 12 through 14. It may be clearer, however, to write them as separate equations.

Lastly we give the relationships between the delete and retrieve character functions. If we start with a non-empty file, it is possible to delete without having inserted (directly) beforehand. The following axioms correspond to equations 10 and 11:

$$\begin{aligned} \text{ch}(\text{del}(F, P), N) &\doteq \text{ch}(F, N+1) \\ &\quad \text{if } (P \leq N) && (15) \\ \text{ch}(\text{del}(F, P), N) &\doteq \text{ch}(F, N) \\ &\quad \text{if } N < P && (16) \end{aligned}$$

We could convert this set of axioms into a program in the equational language OBJ [4]. We also could use the axioms to implement a text editor, which although not very efficient in its performance, is useful in testing whether the specification makes sense. To do so, we just transform these axioms into Horn clauses, as we describe in the next section.

2.3 A Horn Logic Version of the Text Editor

An implementation in a pure logic programming language requires axioms in a certain form, namely Horn clauses. A "clause" is a disjunction of literals (positive or negated "atomic sentences"). A "Horn clause" is a clause with at most one positive literal, which can be written as

$$\neg B_1 \vee \ldots \vee \neg B_s \vee A.$$

This is equivalent to a formula of the form:

$$\forall X_1 \ldots \forall X_n B_1 \wedge \ldots \wedge B_s \supset A$$

where

$$X_1, \ \ldots, X_n$$

are the variables appearing in the formulae

$$B_1, \ldots, B_s \text{ and } A.$$

Because Prolog does not handle functions, we must transform our equations into predicates. For example, for the three-place function "ins(C, F, P)" we introduce the four-place predicate "ins$(C, F, P, Result)$." The predicate is true whenever $Result$ represents the file obtained by inserting character C at position P in file F. We use the same name or "functor" (e.g., "ins") for the three-place function and the four-place predicate, assured that the arity (the number of arguments) will determine which we mean to invoke. It is a simple exercise to transform all the equations in section 2.2 into Horn clauses. We give examples in section 4.

3 Operational Semantics

Now we turn to implementing—developing an operational model for—our logic model. An operational semantics for a given task consists of a set of algorithms to perform it. For text editing, the operational semantics consists of actual procedures to perform *ins*, *del* and *ch*. From a logical point of view, we shall see that an operational semantics is just a model, i.e., a domain of discourse and an interpretation of the function and predicate symbols that satisfies the axioms of the theory.

To back this up we briefly sketch the logical notions of model and structure. For more detail we refer the reader to the literature [6,7]. Given a logic language L, a *structure* S for that language consists of a domain of discourse D_S and interpretations for the function and predicate symbols. If f is a function symbol of n arguments, then the interpretation of f in S, denoted by f_S, is a function taking n-tuples of elements of D_S as arguments. That is, $f_S : D_S^n \mapsto D_S$. Note that a constant c is interpreted as an element $c_S \in D_S$. The interpretation can be extended in a unique way to *ground terms* so that a term t is interpreted by t_S. Ground terms are terms with no variables. This is achieved by the recursive definition $[f(t_1, \ldots, t_n)]_S = f_S([t_1]_S, \ldots, [t_n]_S)$. Predicates are interpreted as subsets so that if P is a predicate of n arguments, then $P_S \subset D_S^n$. A formula $P(t_1, \ldots, t_n)$ holds in (or is *satisfied* by) the language's structure S, written $S \models P(t_1, \ldots, t_n)$, if $(t_1, \ldots, t_n) \in P_S$. The notion of satisfaction for more complex formulae involving logical connectives, variables and quantifiers is defined in a natural way [6,7].

For a set of axioms Γ we say that a structure M is a *model* of the logic theory Γ, written $M \models \Gamma$, if the axioms in Γ are satisfied by the structure M. The notion of a *model* is the method in logic for talking about *truth*. We shall see that the operational semantics for a task should represent a truthful model or implementation of the task as embodied in the axioms of the task's logic theory.

The word *model* has two meanings. Its logical meaning refers to a structure that satisfies a *logic theory*. The other sense of the word *model* refers to describing and representing tasks. Our paper is about the latter use of logic modeling as a tool. That is the meaning of the word *model* in the title of this paper. In this section, however, we shall concentrate on the former meaning of the word because we are discussing the structural relationship between implementations and logic.

An *operational semantics* for a given task consists of a set of algorithms to perform the given task. One can think of a computer program for text editing as an operational semantics for the task of text editing described through the logic model we presented in section 2. It is in this sense that a program becomes an operational semantics for a logic theory. We develop this idea next.

Consider a straightforward implementation of a text editor. We could represent our system in a procedural programming language such as "C". Our data structure for a file could be a C-record consisting of an integer field, `eof`, denoting the length of the file and a string field, `text`, containing the actual characters present in the file:

```
typedef struct
{ int eof;
  char *text; } file;
```

For example, a file with contents `abc` is depicted in Figure 1a.

Place Figure 1a here.

The *character retrieval* operation is quite simple:

```
char ch(F,P)
  file *F;
  int P;
{ if (F→eof <= P)
    ch = EOF_CHAR;
  else ch = F→text[P];}
```

(In this paper's code and clauses we shall be using the abbreviations `F` and `P` for "file" and "position" respectively. These correspond to the variables F and P in section 2. `text[P]` represents the character at position P.)

The function to implement *character insertion* can be realized by the ins C-function.

```
file *ins(C,F,P)
  char C; /* the character to be inserted */
  file *F; /* F is the file where the character C is to be inserted */
  int P; /* the position at which the insertion is to occur.  */
{
  if P > F→eof
    ins = F;
  else
    {
      for i:= P to F→eof do
        F→text[i+1] := F→text[i];
      F→text[P] := C;
      F→eof := F→eof +1
    }
}
```

Similarly, we can write a **C** function del(F, P) to delete the character at position P from file F.

How do operational semantics and logic models relate? The **C** computer program we just described provides a domain of discourse and interpretations for the function symbols. The *ins, del* and *ch* predicates are interpreted by **C** functions. Thus the computer program also is, in some sense, a *structure* for the logic theory. Recall that a structure for a logic theory provides (1) a domain of discourse, i.e., a *universe*, the elements of which correspond to individual objects in the theory and (2) an interpretation for the function symbols, i.e., an actual function for each function symbol in the theory. These functions take elements of the domain of discourse as their arguments and attain their values within the same domain.

The domain of discourse of the **C** computer program described above consists of finite sequences of bits (*strings of bits*) that constitute **C** objects. That is, bit strings correspond to either a **C**-integer, a **C**-character or a **C**-object of type **file** as defined above. The interpretation will map the constant λ to the sequence of strings which corresponds to a **C**-structure of type `file` with `eof` field of value 0 and `text` field containing an empty string. The function symbol *ins* is interpreted by the **C** function `ins`. What do we mean by the **C** function `ins`? We do not refer to the code sequence of bits representing this function in machine language. We refer to the mathematical function realized by this code that maps bit-strings to bit-strings.

So far we have interpreted functions of the logic theory. If we transform these functions into predicates as suggested earlier, we also must provide interpretations for those predicates. We do this by looking at the set of tuples that conform the function. That is, if the function is $f : X \mapsto Y$, the corresponding predicate is interpreted by the set of tuples $\{< x, f(x) > | x \in X\}$.

We have just shown how an operational semantics leads to a structure of the corresponding logic theory. These structures satisfy all the equations in the logic theory and are therefore *models* (in the logic meaning of the word) of the logic theory. This is true whenever the operational semantics represents a truthful implementation of the specifications embodied in the logic theory. The relation between models and operational semantics is thus: given a logic theory, an operational semantics that is correct with respect to the specification of the logic theory is a model of that theory.

This relation provides us with a tool to test the correctness of implementations. We know that if the implementation violates an equation in the logic theory, then it can not be a model and hence it is not a correct implementation of the task specified by the logic theory. We will use this fact in section 5 to develop a computer program to help detect incorrectness of implementations.

Note that not every model lends itself to be programmed because some models might not be computable. Thus, some models fail to represent operational semantics.

In the sections that follow, we shall be implementing our text editor and correctness checks in Prolog, which is a declarative (non-procedural) language. Instead of a record, our data structure of a file is a list containing two arguments, as illustrated in Figure 1b. The first is an integer representing the position of the *eof-char*. The second argument is a list of the text characters in the file followed by a single *eof-char*. The Prolog code for the predicates *ins*, *del* and *ch* is given below. Note that we have had to convert the functions *pred* and *succ* as these are referenced by *ins* and *del*. (In the declaration of `pred`, we see that Prolog implements "or" with multiple instances of a predicate. If one instance is not executed successfully, the next occurrence of the predicate with the same arity is attempted. This continues until one of the occurrences succeeds or until none remains untried.) The empty file is initialized so that `Eof-P` is 0. The end of file marker is represented by the constant `eof-char`.

Place Figure 1b here.

```
pred(0,0).

pred(P,Before-P) :-
   P > 0,
   Before-P is P - 1.

succ(P,After-P) :-
   After-P is P + 1.

ins(Char,[Eof-P,Text],P,[Eof-P,Text]) :-
   Eof-P < P.

ins(Char,[Eof-P,Text],P,[R-Eof-P,R-Text]) :-
   pred(P,Before-P),
   sublist(Text,0,Before-P,PreText),
   sublist(Text,P,Eof-P,PostText),
   append([PreText,Char,PostText],R-Text),
   succ(Eof-P,R-Eof-P).
```

The first instance of `ins` checks that the position `P` is not past the end of file (the last character of the text list). The second separates the text before and after the insertion point, puts the inserted character between and concatenates the three text portions to create the new text `R-Text`. The

Prolog predicate *sublist* retrieves a designated range from a list as a new list. The Prolog predicate *append* concatenates the contents of the input list in the first argument and returns these in the second. The first argument is a list of lists. (Alternatively *append* could take the second argument and split it into the three sublists in the first argument. This dual functionality is a feature of logical axioms.) The predicate declaration concludes by incrementing the end of file position to `R-Eof-P`.

We implement deletion as follows:

```
del([Eof-P,Text],P,[Eof-P,Text]) :-
   Eof-P <= P.

del([Eof-P,Text],P,[R-Eof-P,R-Text]) :-
   pred(P,Before-P),
   sublist(Text,0,Before-P,PreText),
   succ(P,After-P),
   sublist(Text,After-P,Eof-P,PostText),
   append([PreText,PostText],R-Text),
   pred(Eof-P,R-Eof-P).
```

Again, the first instance of `del` checks that we are not deleting past the end of file. The second instance takes the sublists on either side of the character to be deleted and concatenates them to form the new text `R-Text`. The declaration concludes by decrementing the end of file position.

We implement character retrieval with the `getchar` predicate as follows:

```
getchar([Eof-P,Text],P,eof-char) :-
   Eof-P <= P.

getchar([Eof-P,Text],P,Char) :-
   at-position(Text,P,Char).
```

The first instance of `getchar` returns `eof-char` if we retrieve from past the end of file. The second instance uses the predicate *at-position* to retrieve the character at position `P` in the text list. As the definition of *at-position* is straightforward, we omit it here.

Call this operational semantics $\mathcal{O}$. As described at the beginning of this section, we can view $\mathcal{O}$ as a model for the logic theory of section 2.2, which describes the text editing task. The domain of discourse consists of the data structures used in $\mathcal{O}$, namely lists for files, arrays of characters for text, non-negative integers for end of file counter positions, etc. The function symbol *ins* is interpreted by the predicate `ins` which takes a character, a file and a position as arguments and returns a file. Similarly *del* is interpreted by `del` and *ch* by `getchar`. Furthermore, it is easy to show that $\mathcal{O}$ is a model of the set of axioms Γ given in section 2.2, i.e., that it satisfies every axiom. As an example consider axiom 12:

$$\mathrm{del}(\mathrm{ins}(C, F, N), N) \doteq F$$

which converts to Prolog code:

```
:-  ins(C, F, N, F2),
    del(F2, N, F)
```

If position `N` is not after the end of file, then the result in $\mathcal{O}$ of `ins` will be to shift all characters in F one place to the right from position N, returning file $F2$. The result of the `del` is to shift those characters one place to the left returning F again. The combination of both operations leaves F unchanged. The case where $eof(F, N)$ is simpler as `ins` and `del` return the files unchanged and $F \equiv F2$.

In a similar fashion we can check that all other axioms in Γ are satisfied by this Prolog code. Thus $\mathcal{O}$ captures the meaning of our text editor. In a sense this corroborates our intuition that the set of axioms is satisfactory because most programmers would agree that $\mathcal{O}$ performs the operations wanted in a truthful way. This, of course, is not the only possible operational semantics for this model. Any other correct implementation (say, in a different language, with different data structures such as hashing tables or linked lists, or with different algorithms to help do global search faster) also constitutes a model of Γ.

4 A Prolog Implementation of a Text Editor

We have used the Prolog language to build a basic text editor in accordance with the operational semantics $\mathcal{O}$ declared in the previous section. Prolog is not nondeterministic, nor is its choice of goals arbitrary. It predictably performs a sequential depth-first search on its goal by always choosing the first (leftmost) goal it encounters and backtracking when necessary. It is recognized, however, as the most developed language based on logic programming [3] and is both adequate and practical for computer implementations of logic models. These deviations do not affect our specifications.

Our text editor features the following commands:

`ins`: insert a character into a file
`del`: delete a character from a file
`getchar`: retrieve a character
`ins-block`: insert a block of characters
`del-block`: delete a block of characters
`char-block`: retrieve a block of characters
`backspace`: delete the character preceding the current position in the file
`copy`: copy a block of characters to another file
`cut`: move a block of characters to another file
`find`: locate a block of characters without altering
`replace-generic`: replace a block of characters by another block of characters
`replace-related`: remove, transform and replace text

Together these commands can emulate, for example, the complete functionality of Apple Computer's Macintosh text editing routines [9] (not including those concerning memory and screen management).

For each command we first give the logical function. We then show the Horn logic version of its corresponding predicate. For `ins-block` and `del-block` we also give the code for the Prolog implementation of the predicate using the operational semantics described in section 3. These two samples should suffice as the Prolog code is so similar to the Horn logic predicates. (For the sake of clarity, predicate and variable names in the Prolog code shown here have been shorted slightly from those in the code we actually implemented.) After describing the command set we provide two examples.

4.1 Insert Block

With the `insert-block` operation we introduce standard list notation to our language. `[ ]` denotes the empty list of no elements. `[H|T]` denotes the splitting of the list where `H` is the first element (the "head" or "car") of the list and `T` is the remaining list (the "tail" or "cdr") once the first element is removed. `T` may be the empty list.

Axioms

```
ins-block([ ],F,P) ≐ F
ins-block([Bh|Bt],F,P) ≐
   ins-block(Bt,ins(Bh,F,P),P+1)
```

Horn Logic Predicates

```
ins-block([ ],F,P,F)
ins(Bh,F,P,Ins-F) ∧
succ(P,After-P) ∧
ins-block(Bt,Ins-F,After-P,F-Out) ⊃
   ins-block([Bh|Bt],F,P,F-Out)
```

Prolog Implementation

```
ins-block([ ],F,P,F).
ins-block([BH|BT],F,P,F-Out) :-
   ins(BH,F,P,Ins-F),
   succ(P,After-P),
   ins-block(BT,Ins-F,After-P,F-Out).
```

4.2 Delete Block

The axioms for delete-block are found in section 2.2.

Horn Logic Predicates

```
del(F,P,F-Out) ⊃
   del-block(F,P,P,F-Out)
P1 < P2 ∧
del(F,P2,Del-F) ∧
pred(P,Before-P) ∧
```

```
del-block(Del-F,P1,Before-P,F-Out) ⊃
   del-block(F,P1,P2,F-Out)
```

Prolog Implementation

```
del-block(F,P,P,F-Out) :-
   del(F,P,F-Out)
del-block(F,P1,P2,F-Out) :-
   P1 < P2,
   del(F,P2,Del-F),
   pred(P,Before-P),
   del-block(Del-F,P1,Before-P,F-Out).
```

4.3 Char Block

This operation returns a block of characters in list format from a file. It cannot be used to retrieve the *eof-char*.

Axioms

```
char-block(F,P1,P2) ≐ [ ]
   if getchar(F,P1) ≐ eof-char
char-block(F,P1,P2) ≐ [ ]
   if (P2 < P1)
char-block(F,P1,P2) ≐
   [getchar(F,P1)|char-block(F,P1+1,P2)]
   if (P1 < P2)
   if getchar(F,P1) > eof-char
```

Horn Logic Predicates

```
getchar(F,P1,eof-char) ⊃
   char-block(F,P1,P2,[ ])
P2 < P1 ⊃
   char-block(F,P1,P2,[ ])
P1 < P2 ∧
getchar(F,P1,Char) ∧
Char > eof-char ∧
succ(P1,After-P1) ∧
char-block(F,After-P1,P2,Block) ⊃
```

```
   char-block(F,P1,P2,[Char|Block])
```

4.4 Backspace

This operation deletes the character at the position prior to the one indicated. Of course, one may not backspace past the start of the file or over the *eof-char*.

Axioms

```
backspace(F,0) ≐ F
backspace(F,P) ≐ del(F,P-1)
```

Horn Logic Predicates

```
backspace(F,0,F)
pred(P,Before-P) ∧
del(F,Before-P,F-out) ⊃
   backspace(F,P,F-out)
```

4.5 Copy

This operation copies text between files. It returns the updated version of the file to which the text is copied.

Axiom

```
copy(F1,P11,P12,F2,P2) ≐
   ins-block(char-block(F1,P11,P12),F2,P2)
```

Horn Logic Predicate

```
char-block(F1,P11,P12,Block) ∧
ins-block(Block,F2,P2,F2-Out) ⊃
   copy(F1,P11,P12,F2,P2,F2-Out)
```

4.6 Cut

This operation copies text from one file to another and deletes the copied text from the first. It returns a list containing both file F2, to which we copied and file F1, from which we copied. (We could define separate *cut* functions returning the new state of either file.)

Axiom
```
cut(F1,P11,P12,F2,P2) ≐
   [copy(F1,P11,P12,F2,P2),
       del-block(F1,P11,P12)]
```

Horn Logic Predicate
```
copy(F1,P11,P12,F2,P2,F2-Out) ∧
del-block(F1,P11,P12,F1-Out) ⊃
   cut(F1,P11,P12,F2,P2,[F2-Out,F1-Out])
```

4.7 Find

Like `getchar()` and `char-block()`, this operation does not affect the file. Nothing is inserted or deleted. `find` determines whether a specific block of characters is in the file. If so, the start and end position of the block are returned as the list `[P1,P2]`.

Axiom
```
find(F,Block) ≐ [P1,P2]
   if char-block(F,P1,P2) ≐ Block
```

Horn Logic Predicate
```
char-block(F,P1,P2,Block) ⊃
   find(F,Block,[P1,P2])
```

4.8 Replace - Generic

There are two `replace` operations. We represent the first as a function of arity 4 with a corresponding predicate of arity 5. It replaces a block of text with another block of text. The blocks may be of different lengths.

Axiom
```
replace(F1,P1,P2,Block) ≐
  ins-block(Block,del-block(F1,P1,P2),P2)
```

Horn Logic Predicate
```
del-block(F,P1,P2,Delete-F) ∧
```

```
ins-block(Block,Delete-F,P1,F-Out) ⊃
  replace(F1,P1,P2,Block,F-Out)
```

4.9 Replace Related

We represent the second `replace` as a function of arity 3 with a corresponding predicate of arity 4. It takes a block of text, performs operation `op` on it and replaces the original text with the results of that operation. The function `op(Block-In)` $\doteq$ `Block-Out` can be defined by the user. Examples include filtering out non-printable characters, reversing the characters and converting the text to a different format.

Axiom

```
replace(F1,P1,P2) ≐
   ins-block(op(char-block(F,P1,P2)),
       del-block(F,P1,P2),P1)
```

Horn Logic Predicate

```
char-block(F,P1,P2,Block-In) ∧
op(Block-In,Block-Out) ∧
del-block(F,P1,P2,Delete-F) ∧
ins-block(Block-Out,Delete-F,P1,F-Out) ⊃
   replace(F1,P1,P2,F-Out)
```

4.10 Examples

Figure 2a illustrates invoking the following operations in our Prolog implementation of a text editor, which we built using the operational semantics from this and the previous sections.

$$ins(a, (ins(c, ins(b, \lambda, 0), 1), 0)$$

$$ins(a, (ins(b, ins(c, \lambda, 0), 0), 0)$$

$$del(ins(b, ins(c, ins(x, ins(a, \lambda, 0), 1), 2), 2), 1)$$

Each of these operations results in the identical file. This is similar in nature to the example at the end of section 2.2.

Place Figure 2a here.

Figure 2b contains a longer example in which we create the text "Sample Sentence " by using many of the operations from this section. In this example, *replace-related* reverses the text in the block chosen.

Place Figure 2b here.

5 Testing the Correctness of an Existing Text Editor

In the previous section we used the logic model and operational semantics developed in sections 2 and 3 to design and implement a text editor. In this section we use them to test the correctness of an existing implementation of a text editing system. We chose to examine a base subset of a widely-used set of system routines—the internal text editing "toolbox" routines incorporated in the operating system of the Macintosh computer. This demonstrates that our axioms can be applied outside of the original system but within their intended domain.

We use the Hoare logic approach [11] to ensure the correctness of every toolbox editing routine by establishing and checking pre- and post-conditions. Operations can be defined and characterized by their input and output constraints (logical rules and properties). If the system is in a legal state and only legal operations are executed, then the data of the resulting system will be guaranteed to have semantic integrity. Hoare's use of pre- and post-conditions is important for three reasons: (1) the checking of particular constraints is tied to particular operations, (2) the pre- and post-constraints for the operation act as a guide for the implementation, and (3) they are used to prove that the operation's specification and implementation are correct [12]. Before the operating system executes each toolbox routine, we determine the current system state. This is our pre-condition. After each routine we ensure that the state of the system reflects only the changes we expect due to the execution of that routine. This is our post-condition.

Place Figure 3 here.

We use the `perform` predicate to implement this in Prolog. We "sandwich" the pre- and post- conditions around the Macintosh toolbox routines and observe as users invoke each routine from the interface. Users invoke text editing commands from keystrokes (such as typing a character or pressing the backspace key) or menu commands. Figure 3 illustrates our testing environment. The Macintosh "event loop" traps each text editing command. Our system examines the user command and instead of passing it directly to the toolbox, passes it to the `perform` predicate with the name of the actual toolbox routine as an additional parameter. First, `perform` records the state of the Macintosh file before the user's command. It then executes the actual toolbox call associated with the command and records the actual state of the Macintosh file resulting from the execution of the routine. Next it simulates the execution of the toolbox routine using the predicates developed in section 4. If the simulation produces the same results as executing the actual toolbox routine, then we consider the toolbox routine to be correct. Otherwise the routine is flagged as not being correct. We code the `perform` predicate as follows:

```
perform(SimOp, UserInput, ActualCmd, File-ID) :-
  before-operation(File-ID, F-Before, P1, P2),
  call(ActualCmd, File-ID),
  after-operation(File-ID, F-After),
  simulate(SimOp, F-Before, P1, P2, F-After),
  message(UserInput, P1, P2, SimOp, Flag, File-ID).
```

`SimOp`: the Prolog predicate that will simulate the toolbox command
`UserInput`: the action the user chose to invoke the toolbox command
`ActualCmd`: the actual toolbox command that implements the user's action
`File-ID`: an internal pointer to the actual file stored in computer memory
`F-Before`: a simulation of the file's state before executing the toolbox command
`F-After`: a simulation of the file's state after executing the toolbox command
`P1`: the starting character position of the current selection
`P2` the ending character position of the current selection
`Flag`: "ok" if the simulation matches the actual file; "not ok" otherwise

A full listing of the **perform** predicate and how we intercepted the event loop can be found in Appendix 2.

We found we had to account for several characteristics of the Macintosh text editing environment. Character positions actually are marked between characters. For example, if you want to select the first character in a file you must specify from position 0 to position 1. Thus it is possible to have a null selection. Also, several commands have different functions depending on whether the user has made a null selection or a non-empty selection. For example, when the user presses the backspace key with a null selection, the toolbox performs the equivalent of a backspace command, otherwise it performs the equivalent of a *del-block*. Another difference is that the Macintosh file has no *eof-char*. We simulate it based on the length of the text for testing.

We have tested the following Macintosh toolbox text editing commands (many of which invoke more that one operation depending on the user action and the current selection) thoroughly in the environment shown in Figure 3. We present the full test in Appendix 1. We are hereby pleased to announce that all of the toolbox text editing commands are correct according to the axioms of the logic model we have developed in this paper!

tekey (*insert/backspace*) - insert/backspace/replace-generic/delete-block

tecut (*cut*) - cut a character block to the clipboard file

tecopy (*copy*) - copy a character block to the clipboard file `tepaste` (*paste*) - insert/replace-generic a character block from the clipboard file

`tedelete` (*clear*) - delete-block

`munger` (*find*) - determine the position of a block of characters

6 A Larger Application of Logic Modeling

In this section we discuss how we are applying these logic modeling techniques to a larger information system.

We are part of a research group developing a decision support system (DSS) shell to support individual applications that declare their components to a knowledge-base [17]. The shell will comprise a model management subsystem "TEFA" [14,10], a scenario and data management subsystem "OLEOS," a user interface subsystem "Maxi" [15,16], and communication links among these. Together these subsystems provide an environment in which an application builder has to specify just the models, data and report formats that are unique to his or her application. The shell handles all sharing, retrieval and execution functions for models and data, as well as communications with the end user. We are modeling each of these subsystems and communication methods in logic before implementing it. Our goal is to make it cheaper and easier for application builders to develop DSS applications, and for users such as decision analysts to work with them.

To illustrate our use of logic modeling we shall focus here on a single aspect of the user interface subsystem, providing hypertext-style navigation as a "system-level" feature. The shell automatically generates decision reports known as "interactive documents" such as the hypothetical ones in Figure 4. It automatically infers which elements of the application builder's knowledge base map to hypertext "nodes", "links" and "buttons". Nodes are the objects of interest such as particular decision models and data. Links represent relations among these, e.g., which models use which data, as well as DSS commands, e.g., describing and executing a model. We highlight hypertext buttons in boldface within interactive documents to indicate links (e.g., the name of a model to execute, a numerical result to query). It is through these buttons that users navigate or "traverse" around the DSS applications.

Place Figure 4 here.

In building the interface we are using logic modeling in two distinct areas: to manage hypertext buttons in the text editor, and to control the inference of hypertext nodes and links in preparation for and during link traversal.

We have expanded the standard functionality of a text editor based on our model from section 2 to account for additional hypertext-specific conditions. For example, interactive documents can contain several types of buttons, each of which must be validated against user actions in a different way. Some, such as the execution results shown in Figure 4, may be deleted by the user but not modified. Others, such as keyword buttons, may be modified, but doing so invalidates the buttons and the system must deregister them. In addition to validation, the character positions of all buttons after the modification point must be updated whenever text is added or deleted. We have modeled validation and positioning with logical axioms.

Now to the inference of nodes, links and buttons for link traversal. (For more details see [18].) We believe that application builders should not have to express the contents of their knowledge base—mathematical models and data, etc.—in terms of hypertext nodes, links, buttons. The shell (Maxi and TEFA) provides a set of general logical axioms called "bridge laws" that map components of application knowledge bases to hypertext entities. To do so, bridge laws employ the known structure of the shell's DSS application components.

For example, we declare nodes as follows:

$$\forall A \forall N \text{node}(N, A)$$

where N is the identifier of the node and A is a list of arbitrary attributes which are available for inferencing. The declaration has been simplified for this presentation—see [16,18] for more details. The following bridge law maps all of an application's mathematical models to hypertext nodes:

$$\begin{array}{l} \forall X, S, D\ model(X) \wedge \\ type(X, mathematical\_model) \wedge \\ source(X, S) \wedge \\ description(X, D) \ \supset node(X, [origin(S), definition(D)]) \end{array}$$

where X is the model identifier, S is its source and D is a definition for it. With this one axiom, all mathematical models in any application's knowledge base compatible with the DSS shell are now accessible as hypertext entities.

Logic modeling has helped us in specifying, verifying and prototyping these aspects of the user interface. It has been most beneficial in designing the other subsystems as well.

7 Conclusion

It is well known that maintenance costs commandeer up to 75% of the life cycle cost of software systems. Almost all new information systems contain "bugs" and periodically systems need to be upgraded to accommodate new business conditions (e.g., new tax laws) [13]. Logic modeling can play an important role in the planning and testing of computer systems. Systems analysts could employ it as one of their standard design tools to minimize the potential for design flaws. Logic modeling forces the analyst to detail the system's features and components clearly and unambiguously. The logic model also can serve as documentation and as a base for designing system enhancements. Logic modeling is useful for testing correctness of both designs and implementations. Given the inevitability of bugs as well as the recent threat of viruses, people responsible for testing implementations could turn to the methods developed in section 5 as one weapon in their testing arsenal.

In this paper we have shown how to represent the essential aspects of a text editor in a logic model. This exercise resulted in a simple first order theory which we then used for two purposes:

1. to construct a prototype implementation in Prolog, and
2. to test another implementation of a text editor (the internal Macintosh toolbox editing functions).

We feel strongly that this two-fold use of logic modeling constitutes an important contribution to the area of software development. Logic modeling naturally coordinates

1. software specification (via Horn logic),
2. prototypical implementations, and
3. developing a testing strategy to evaluate implementations.

These aspects of logic modeling clearly support its use as a software development tool.

8 Acknowledgements

We wish to thank Hemant Bhargava of the Naval Postgraduate School, Chris Jones of Simon Fraser University and Steven O. Kimbrough of the Wharton School for their invaluable help with this project. This work was supported in part by the U. S. Coast Guard, under contract DTCG39-86-C-E92204 (formerly DTCG39-86-C-80348), Steven O. Kimbrough principal investigator.

This paper expands an earlier version appearing as [19].

Version 1.9 -November 21, 1991.

References

[1] Kimbrough, Steven O. and Ronald M. Lee, "Logic Modeling: A Tool for Management Science," *Decision Support Systems*, vol. 4, p. 3-16, 1988.

[2] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer Verlag, Berlin 1980.

[3] Leon Sterling and Ehud Shapiro, *The Art of Prolog: Advanced Programming Techniques,* MIT Press, Cambridge, 1986.

[4] K. Futatsugi, Joseph Goguen, J. Juoannoud, and José Mesegue, "Principles of OBJ2," *Proceedings, 1985 Symposium on Principles of Programming Languages*, p. 52–66, ACM, 1985.

[5] Frederick Hayes-Roth, "Rule Based Systems," *Communications of the ACM*, vol. 28, no. 9, p. 921- 932, September 1985.

[6] Jean H. Gallier, *Logic for Computer Science*, Harper and Row, 1985.

[7] Herbert B. Enderton, *A Mathematical Introduction to Logic*, New York: Academy Press, 1972.

[8] José Meseguer and Joseph Goguen, "Initiality, Induction and Computability," in M. Nivat and J. Reynolds, eds., *Algebraic Methods in Semantics*, p. 459–540, Cambridge University Pess, 1985.

[9] Apple Computer, Inc., *Inside Macintosh Volume I*, p. 393-394, 1985.

[10] Hemant Bhargava, *A Logic Model for Model Management: An Embedded Languages Approach,* Ph. D. Thesis, Decision Sciences Department, University of Pennsylvania, Philadelphia, 1990.

[11] K. R. Apt, "Ten Years of Hoare's Logic: A Survey - Part 1," *ACM TOPLAS*, vol. 3 no. 4 p. 431-483, October 1981.

[12] Anthony I. Wasserman, *et al.*, "Developing Interactive Information Systems with the User Software Engineering Methodology," in R. Becker and W. Buxton, eds., *Readings in Human- Computer Interfaces: A Multidisciplinary Approach*, p. 508-527, Morgan Kaufmann Publishers, Inc., 1985.

[13] James C. Emery, *Management Information Systems: The Strategic Critical Resource*, Oxford Press, New York, 1987.

[14] Hemant K. Bhargava and Steven O. Kimbrough, "On Embedded Languages for Model Management," *Proceedings of the Twenty-Third Hawaii International Conference on System Sciences*, IEEE Press, Washington, D.C., 1990.

[15] Michael Bieber and Steven O. Kimbrough, "On Generalizing the Concept of Hypertext," forthcoming in *Management Information Systems Quarterly.*

[16] Michael Bieber, *Generalized Hypertext in a Knowledge-based DSS Shell Environment,* Ph. D. Thesis, Decision Sciences Department, University of Pennsylvania, Philadelphia, 1990.

[17] Steven O. Kimbrough, Clark Pritchett, Michael Bieber and Hemant Bhargava, "The Coast Guard's KSS Project," *Interfaces*, vol. 20 no. 6 p. 5-16, November 1990.

[18] Michael Bieber and Steven O. Kimbrough, "On the Logic of Generalized Hypertext," *in this issue of Decision Support Systems.*

[19] Michael Bieber and Tomás Isakowitz, "A Logic Model for Text Editing," *Proceedings of the Twenty-Second Hawaii International Conference on System Sciences,* IEEE Press, Washington, D.C., 1989.

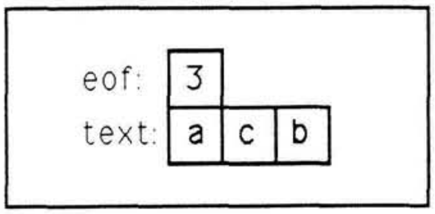


Figure 1a - Our "record" data structure for a text editing file implemented in "C", where the file contains the text *acb*.

```
[3, [a, c, b, eof-char]]
```

Figure 1b - Our "list" data structure for a text editing file implemented in Prolog, where the file contains the text *acb*.

```
?- Empty_File = [0,[eof_char]],
ins(b,Empty_File,0,F0),
ins(c,F0,1,F1),
ins(a,F1,0,Resulting_File).
        Resulting_File = [3,[a,b,c,eof_char]]

?- Empty_File = [0,[eof_char]],
ins(c,Empty_File,0,F0),
ins(b,F0,0,F1),
ins(a,F1,0,Resulting_File).
        Resulting_File = [3,[a,b,c,eof_char]]

?- Empty_File = [0,[eof_char]],
ins(a,Empty_File,0,F0),
ins(x,F0,1,F1),
ins(c,F1,2,F2),
ins(b,F2,2,F3),
del(F3,1,Resulting_File).
        Resulting_File = [3,[a,b,c,eof_char]]
```

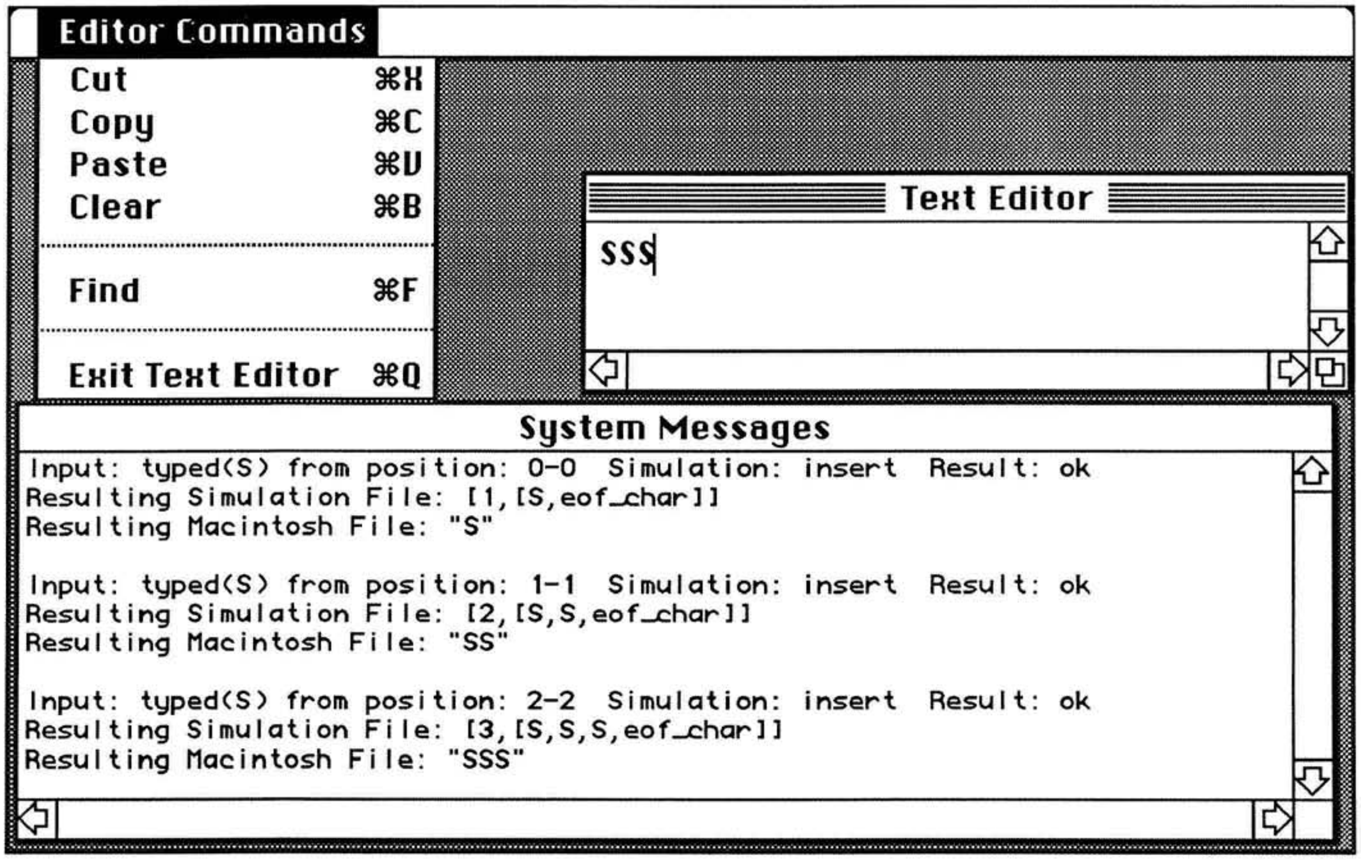
Figure 2a - 3 ways to produce "abc"

```
?- Clip = [0,[eof_char]],                            % define the clipboard file
Empty_File = [0,[eof_char]],                         % define the empty file
ins('S',Empty_File,0,F0),
ins('S',F0,1,F1),
ins_block([e,n,t,t,t,n,e,s,' ','S',a,m,x,p,l,e],F1,2,F2),
backspace(F2,1,F3),                                  % remove first "S"
backspace(F3,0,F4),                                  % no change
del(F4,13,F5),                                       % remove "x"
del(F5,16,F6),                                       % delete "eof_char": no change
del_block(F6,4,5,F7),                                % delete "tt"
replace(F7,6,6,[c,e],F8),                            % replace "s" by "ce"
replace(F8,4,5,F9),                                  % reverse "ne" to "en"
find(F9,['S',a,m,p,l,e],[F9_posn1,F9_posn2]),        % find text
cut(F9,F9_posn1,F9_posn2,Clip,0,[ClipX,F10]),        % cut text to the clipboard file
copy(ClipX,0,10,F10,0,F11),                          % copy whole clipboard to start of file
ins(' ',F11,6,F12).

F0 = [1,['S',eof_char]],
F1 = [2,['S','S',eof_char]]
F2 = [18,['S','S',e,n,t,t,t,n,e,s,' ','S',a,m,x,p,l,e,eof_char]]
F3 = [17,['S',e,n,t,t,t,n,e,s,' ','S',a,m,x,p,l,e,eof_char]]
F4 = [17,['S',e,n,t,t,t,n,e,s,' ','S',a,m,x,p,l,e,eof_char]]
F5 = [16,['S',e,n,t,t,t,n,e,s,' ','S',a,m,p,l,e,eof_char]]
F6 = [16,['S',e,n,t,t,t,n,e,s,' ','S',a,m,p,l,e,eof_char]]
F7 = [14,['S',e,n,t,n,e,s,' ','S',a,m,p,l,e,eof_char]]
F8 = [15,['S',e,n,t,n,e,c,e,' ','S',a,m,p,l,e,eof_char]]
F9 = [15,['S',e,n,t,e,n,c,e,' ','S',a,m,p,l,e,eof_char]]
F10 = [9,['S',e,n,t,e,n,c,e,' ',eof_char]]
ClipX = [6,['S',a,m,p,l,e,eof_char]]
F11 = [15,['S',a,m,p,l,e,'S',e,n,t,e,n,c,e,' ',eof_char]]
F12 = [16,['S',a,m,p,l,e,' ','S',e,n,t,e,n,c,e,' ',eof_char]]
```

Figure 2b - creating the text "Sample Sentence "

This figure shows two snapshots of the Macintosh testing environment we describe in section 5. Users invoke text editing commands implicitly from the keyboard or explicitly from the "pull-down" menu (shown in the first snapshot). The action log displaying our simulation results is in the "System Messages" window. The "Text Editor" window shows the results of the last action shown in the "System Messages" window. (The full example testing each of the Macintosh toolbox routines is in Appendix 1. The main Prolog code implementing this environment is in Appendix 2.)

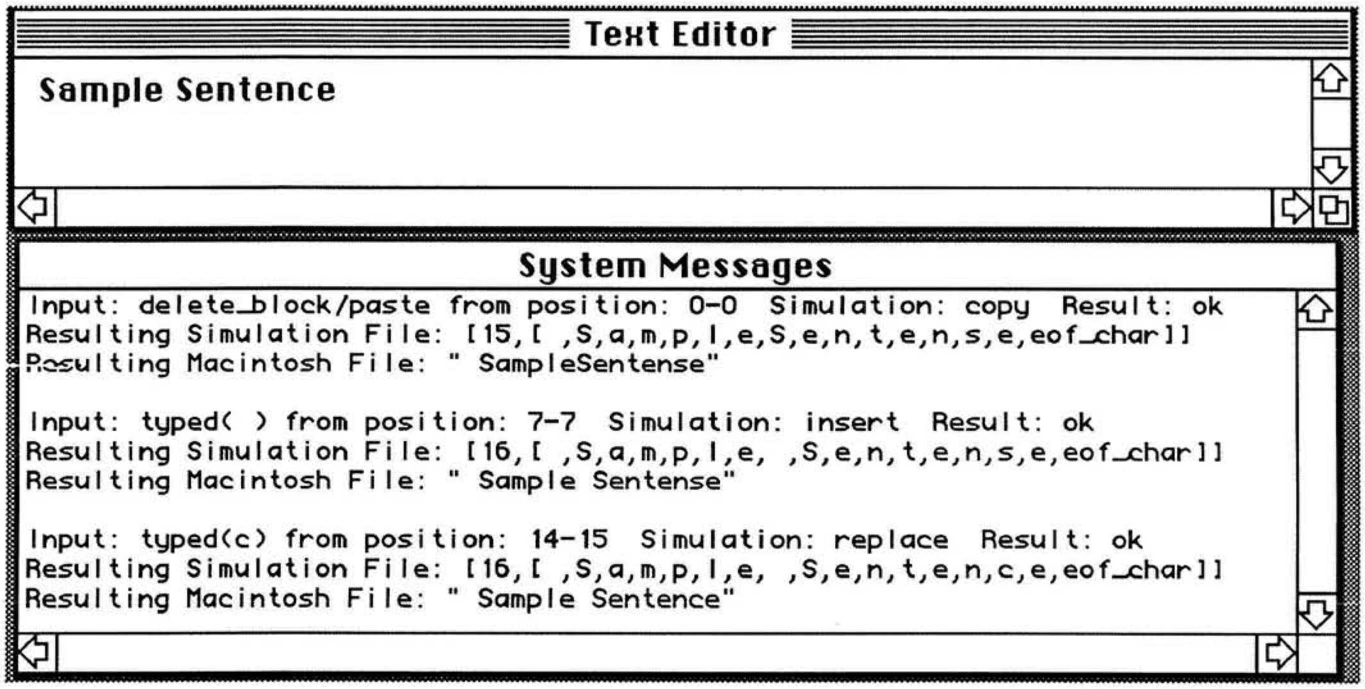


```
Input: typed(S) from position: 0-0  Simulation: insert  Result: ok
Resulting Simulation File: [1,[S,eof_char]]
Resulting Macintosh File: "S"

Input: typed(S) from position: 1-1  Simulation: insert  Result: ok
Resulting Simulation File: [2,[S,S,eof_char]]
Resulting Macintosh File: "SS"

Input: typed(S) from position: 2-2  Simulation: insert  Result: ok
Resulting Simulation File: [3,[S,S,S,eof_char]]
Resulting Macintosh File: "SSS"
```

...

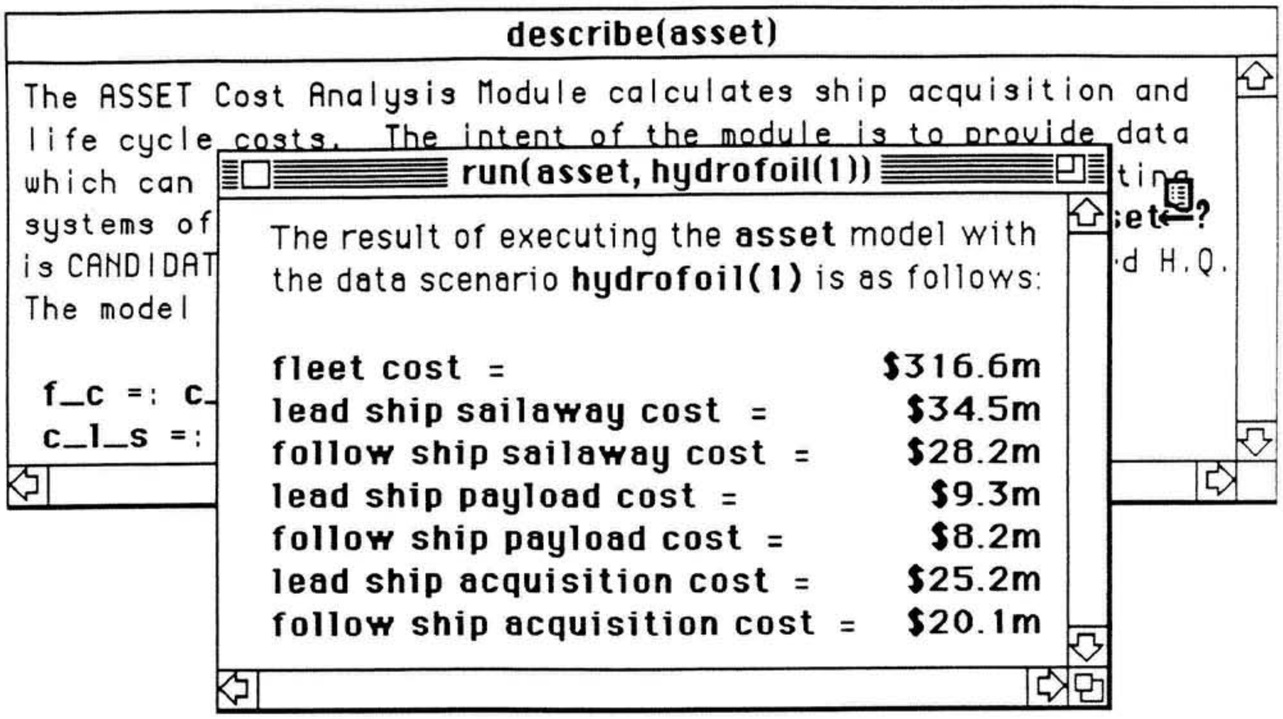


```
Input: delete_block/paste from position: 0-0  Simulation: copy  Result: ok
Resulting Simulation File: [15,[ ,S,a,m,p,l,e,S,e,n,t,e,n,s,e,eof_char]]
Resulting Macintosh File: " SampleSentense"

Input: typed( ) from position: 7-7  Simulation: insert  Result: ok
Resulting Simulation File: [16,[ ,S,a,m,p,l,e, ,S,e,n,t,e,n,s,e,eof_char]]
Resulting Macintosh File: " Sample Sentense"

Input: typed(c) from position: 14-15  Simulation: replace  Result: ok
Resulting Simulation File: [16,[ ,S,a,m,p,l,e, ,S,e,n,t,e,n,c,e,eof_char]]
Resulting Macintosh File: " Sample Sentence"
```

Figure 3 - The Testing Environment on the Macintosh



describe(asset)

The ASSET Cost Analysis Module calculates ship acquisition and
life cycle costs. The intent of the module is to provide data
which can
systems of
is CANDIDAT
The model

f_c =: **c_**
c_l_s =:

run(asset, hydrofoil(1))

The result of executing the **asset** model with
the data scenario **hydrofoil(1)** is as follows:

```
fleet cost =                         $316.6m
lead ship sailaway cost =             $34.5m
follow ship sailaway cost =           $28.2m
lead ship payload cost =               $9.3m
follow ship payload cost =             $8.2m
lead ship acquisition cost =          $25.2m
follow ship acquisition cost =        $20.1m
```

Figure 4 - Two Interactive Documents with Inferred Hypertext Buttons in Boldface

Appendix 1: The Full Example from Figure 3

In this example, we observe a user invoking each of the Macintosh text editing routines described in section 5. This is the text of the transaction log resulting from the user's keystrokes and menu commands, which appears in the "System Messages" window. Figure 3 shows the actual interface during the first and the last three user actions. We present the code underlying this example in Appendix 2.

```
Input: typed(S) from position: 0-0  Simulation: insert  Result: ok
Resulting Simulation File: [1,[S,eof_char]]
Resulting Macintosh File: "S"

Input: typed(S) from position: 1-1  Simulation: insert  Result: ok
Resulting Simulation File: [2,[S,S,eof_char]]
Resulting Macintosh File: "SS"

Input: typed(S) from position: 2-2  Simulation: insert  Result: ok
Resulting Simulation File: [3,[S,S,S,eof_char]]
Resulting Macintosh File: "SSS"

Input: typed(e) from position: 3-3  Simulation: insert  Result: ok
Resulting Simulation File: [4,[S,S,S,e,eof_char]]
Resulting Macintosh File: "SSSe"

Input: typed(t) from position: 4-4  Simulation: insert  Result: ok
Resulting Simulation File: [5,[S,S,S,e,t,eof_char]]
Resulting Macintosh File: "SSSet"

Input: typed(t) from position: 5-5  Simulation: insert  Result: ok
Resulting Simulation File: [6,[S,S,S,e,t,t,eof_char]]
Resulting Macintosh File: "SSSett"

Input: typed(t) from position: 6-6  Simulation: insert  Result: ok
Resulting Simulation File: [7,[S,S,S,e,t,t,t,eof_char]]
Resulting Macintosh File: "SSSettt"

Input: typed(e) from position: 7-7  Simulation: insert  Result: ok
Resulting Simulation File: [8,[S,S,S,e,t,t,t,e,eof_char]]
Resulting Macintosh File: "SSSettte"

Input: typed(n) from position: 8-8  Simulation: insert  Result: ok
Resulting Simulation File: [9,[S,S,S,e,t,t,t,e,n,eof_char]]
Resulting Macintosh File: "SSSettten"

Input: typed(s) from position: 9-9  Simulation: insert  Result: ok
Resulting Simulation File: [10,[S,S,S,e,t,t,t,e,n,s,eof_char]]
Resulting Macintosh File: "SSSettens"

Input: typed(e) from position: 10-10  Simulation: insert  Result: ok
Resulting Simulation File: [11,[S,S,S,e,t,t,t,e,n,s,e,eof_char]]
Resulting Macintosh File: "SSSetttense"

Input: typed( ) from position: 11-11  Simulation: insert  Result: ok
Resulting Simulation File: [12,[S,S,S,e,t,t,t,e,n,s,e, ,eof_char]]
Resulting Macintosh File: "SSSetttense "
```

```
Input: typed(S) from position: 12-12  Simulation: insert  Result: ok
Resulting Simulation File: [13,[S,S,S,e,t,t,t,e,n,s,e, ,S,eof_char]]
Resulting Macintosh File: "SSSetttense S"

Input: typed(a) from position: 13-13  Simulation: insert  Result: ok
Resulting Simulation File: [14,[S,S,S,e,t,t,t,e,n,s,e, ,S,a,eof_char]]
Resulting Macintosh File: "SSSetttense Sa"

Input: typed(m) from position: 14-14  Simulation: insert  Result: ok
Resulting Simulation File: [15,[S,S,S,e,t,t,t,e,n,s,e, ,S,a,m,eof_char]]
Resulting Macintosh File: "SSSetttense Sam"

Input: typed(m) from position: 15-15  Simulation: insert  Result: ok
Resulting Simulation File: [16,[S,S,S,e,t,t,t,e,n,s,e, ,S,a,m,m,eof_char]]
Resulting Macintosh File: "SSSetttense Samm"

Input: typed(p) from position: 16-16  Simulation: insert  Result: ok
Resulting Simulation File: [17,[S,S,S,e,t,t,t,e,n,s,e, ,S,a,m,m,p,eof_char]]
Resulting Macintosh File: "SSSetttense Sammp"

Input: typed(l) from position: 17-17  Simulation: insert  Result: ok
Resulting Simulation File: [18,[S,S,S,e,t,t,t,e,n,s,e, ,S,a,m,m,p,l,eof_char]]
Resulting Macintosh File: "SSSetttense Sammpl"

Input: typed(e) from position: 18-18  Simulation: insert  Result: ok
Resulting Simulation File: [19,[S,S,S,e,t,t,t,e,n,s,e, ,S,a,m,m,p,l,e,eof_char]]
Resulting Macintosh File: "SSSetttense Sammple"

Input: backspace_key from position: 1-1  Simulation: backspace  Result: ok
Resulting Simulation File: [18,[S,S,e,t,t,t,e,n,s,e, ,S,a,m,m,p,l,eof_char]]
Resulting Macintosh File: "SSSetttense Sammpl"

Input: backspace_key from position: 0-0  Simulation: backspace  Result: ok
Resulting Simulation File: [18,[S,S,S,e,t,t,t,e,n,s,e, ,S,a,m,m,p,l,eof_char]]
Resulting Macintosh File: "SSSetttense Sammpl"

Input: clear from position: 4-6  Simulation: delete_block  Result: ok
Resulting Simulation File: [16,[S,S,e,t,e,n,s,e, ,S,a,m,m,p,l,e,eof_char]]
Resulting Macintosh File: "SSetense Sammple"

Input: copy from position: 5-6  Simulation: copy  Result: ok
Resulting Simulation File: [16,[S,S,e,t,e,n,s,e, ,S,a,m,m,p,l,e,eof_char]]
Resulting Macintosh File: "SSetense Sammple"

Input: paste from position: 3-3  Simulation: copy  Result: ok
Resulting Simulation File: [17,[S,S,e,n,t,e,n,s,e, ,S,a,m,m,p,l,e,eof_char]]
Resulting Macintosh File: "SSentense Sammple"

Input: backspace_key from position: 13-14  Simulation: delete_block  Result: ok
Resulting Simulation File: [16,[S,S,e,n,t,e,n,s,e, ,S,a,m,p,l,e,eof_char]]
Resulting Macintosh File: "SSentense Smmple"

Input: find from position: 13-13  Simulation: find  Result: ok
Resulting Simulation File: [16,[S,S,e,n,t,e,n,s,e, ,S,a,m,p,l,e,eof_char]]
Resulting Macintosh File: "SSentense Sample"

Input: cut from position: 9-16  Simulation: cut  Result: ok
Resulting Simulation File: [9,[S,S,e,n,t,e,n,s,e,eof_char]]
Resulting Macintosh File: "SSentense"
```

```
Input: delete_block/paste from position: 0-0  Simulation: copy  Result: ok
Resulting Simulation File: [15, [ ,S,a,m,p,l,e,S,e,n,t,e,n,s,e,eof_char]]
Resulting Macintosh File: " SampleSentense"

Input: typed( ) from position: 7-7  Simulation: insert  Result: ok
Resulting Simulation File: [16, [ ,S,a,m,p,l,e, ,S,e,n,t,e,n,s,e,eof_char]]
Resulting Macintosh File: " Sample Sentense"

Input: typed(c) from position: 14-15  Simulation: replace  Result: ok
Resulting Simulation File: [16, [ ,S,a,m,p,l,e, ,S,e,n,t,e,n,c,e,eof_char]]
Resulting Macintosh File: " Sample Sentence"
```

Appendix 2: The Code of our Macintosh Text Editor Implementation

This is the primary Prolog code implementing the text editor described in section 5, and illustrated in Figure 3 and Appendix 1. The editor intercepts both keyboard inputs and menu selections. The *perform/4* predicate utilizes Hoare logic by establishing pre- and post-conditions (with the predicates *before_operation/4* and *after_operation/2)* that verify the Macintosh text editing routines (the third argument of *perform/4*).

% *KEYBOARD INPUT:*

```
doTyping(FileID,AsciiCode) :-
   determine_character(Char,AsciiCode),      % what did they type?
   mempeek(FileID,32,SelStart),              % selection starting character position
   mempeek(FileID,34,SelEnd),                % selection ending character position
   (SelStart = SelEnd,                       % click (null selection)?
   ( AsciiCode = 8,                          % "8" = backspace key
      perform(backspace,backspace_key,tekey(AsciiCode,FileID),FileID);
      perform(insert,Char,typed(Char),tekey(AsciiCode,FileID),FileID));
    AsciiCode = 8,
   perform(delete_block,backspace_key,tekey(AsciiCode,FileID),FileID);
      perform(replace,[Char],typed(Char),tekey(AsciiCode,FileID),FileID)).
```

% *EDITOR MENU COMMANDS:*

% CUT

```
menu_selection(FileID,1) :-
   perform(cut,cut,tecut(FileID),FileID).
```

% COPY

```
menu_selection(FileID,2) :-
   perform(copy,copy,tecopy(FileID),FileID).
```

% PASTE

```
menu_selection(FileID,3) :-
   mempeek(FileID,32,SelStart),              % get selection start position
   mempeek(FileID,34,SelEnd),                % get selection end position
   (SelStart = SelEnd,                       % click (null selection)?
      perform(paste_insert,paste,tepaste(FileID),FileID);
      perform(paste_replace,'delete_block/paste',tepaste(FileID),FileID)).
```

% CLEAR

```
menu_selection(FileID,4) :-
   perform(delete_block,clear,tedelete(FileID),FileID).
```

% FIND

```
menu_selection(FileID,6) :-
   get_search_text_from_user(TextString),
   perform(find,find,munger(TextString,Result),FileID).
```

% EXIT

```
menu_selection(FileID,8) :-
   retractall(programFinishCondition(_)),
   asserta(programFinishCondition(true)).
```

% PERFORM PREDICATE

```
perform(backspace,MacOperation,Command,FileID) :-
   before_operation(FileID,File_Before,Posn1,Posn2),
   Command,
   after_operation(FileID,File_After),
   simulate(backspace(File_Before,Posn1,File_After),ConditionFlag),
   message(MacOperation,Posn1,Posn2,backspace,ConditionFlag,FileID).

perform(insert,Char,MacOperation,Command,FileID) :-
   before_operation(FileID,File_Before,Posn1,Posn2),
   Command,
   after_operation(FileID,File_After),
   simulate(insert(Char,File_Before,Posn1,File_After),ConditionFlag),
   message(MacOperation,Posn1,Posn2,insert,ConditionFlag,FileID).

perform(delete_block,MacOperation,Command,FileID) :-
   before_operation(FileID,File_Before,Posn1,Posn2),
   Command,
   after_operation(FileID,File_After),
   simulate(delete_block(File_Before,Posn1,Posn2,File_After),ConditionFlag),
   message(MacOperation,Posn1,Posn2,delete_block,ConditionFlag,FileID).

perform(replace,Block,MacOperation,Command,FileID) :-
   before_operation(FileID,File_Before,Posn1,Posn2),
   Command,
   after_operation(FileID,File_After),
   simulate(replace(File_Before,Posn1,Posn2,Block,File_After),ConditionFlag),
   message(MacOperation,Posn1,Posn2,replace,ConditionFlag,FileID).

perform(find,MacOperation,Command,FileID) :-
   before_operation(FileID,File_Before,_,_),
   Command,
   after_operation(FileID,File_After,Posn1,Posn2),
   simulate(find(File_Before,Block,[Posn1,Posn2]),
      find(File_After,Block,[Posn1,Posn2]),ConditionFlag),
   message(MacOperation,Posn1,Posn2,find,ConditionFlag,FileID).

perform(cut,MacOperation,Command,FileID) :-
   before_operation(FileID,File_Before,Posn1,Posn2),
   Command,
   after_operation(FileID,File_After),
   simulate(cut(File_Before,Posn1,Posn2,empty_file,0,[Clipboard,File_After]),
      asserta(clipboard(Clipboard)),ConditionFlag),
   message(MacOperation,Posn1,Posn2,cut,ConditionFlag,FileID).

perform(paste_insert,MacOperation,Command,FileID) :-
   before_operation(FileID,File_Before,Posn1,Posn2),
   Command,
   after_operation(FileID,File_After),
   simulate(clipboard(Clipboard),
      copy(Clipboard,0,10,File_Before,Posn1,File_After),ConditionFlag),
   message(MacOperation,Posn1,Posn2,copy,ConditionFlag,FileID).

perform(paste_replace,MacOperation,Command,FileID) :-
   before_operation(FileID,File_Before,Posn1,Posn2),
   Command,
   after_operation(FileID,File_After),
   simulate(clipboard(Clipboard),
      delete_block(File_Before,Posn1,Posn2,File_Temp),
            copy(Clipboard,0,10,File_Temp,Posn1,File_After),ConditionFlag),
   message(MacOperation,Posn1,Posn2,copy,ConditionFlag,FileID).
```

```
perform(copy,MacOperation,Command,FileID) :-
   before_operation(FileID,File_Before,Posn1,Posn2),
   Command,
   after_operation(FileID,File_After),
   simulate(copy(File_Before,Posn1,Posn2,empty_file,0,File_After),
      copy(File_After,Posn1,Posn2,[0,[eof_char]],0,Clipboard),
              asserta(clipboard(Clipboard)),ConditionFlag),
   message(MacOperation,Posn1,Posn2,copy,ConditionFlag,FileID).



before_operation(FileID,File,Posn1,Posn2) :-
   create_substitute_prolog_file(FileID,File),
   get_current_positions(FileID,Posn1,Posn2).


after_operation(FileID,File) :-
   create_substitute_prolog_file(FileID,File).

after_operation(FileID,File,Posn1,Posn2) :-
   create_substitute_prolog_file(FileID,File),
   get_current_positions(FileID,Posn1,Posn2).


simulate(PrologCommand,ok) :-
   PrologCommand.
simulate(_,'not ok').

simulate(PrologCommand1,PrologCommand2,ok) :-
   PrologCommand1,
   PrologCommand2.
simulate(_,_,'not ok').


create_substitute_prolog_file(FileID,[Length,Text]) :-
   tegettext(FileID,StrPtr),                  % retrieve the file's text
   mempeek(FileID,60,Length),                 % get the file length
   pointer_to_list(StrPtr,Length,TextList),   % convert the file's text to a list of characters
   append(TextList,[eof_char],Text).          % add the eof_char to the simulated file's text


get_current_positions(FileID,SelStart,Posn2) :-   % Positions are marked between characters
   mempeek(FileID,32,SelStart),               % get selection start position
   mempeek(FileID,34,SelEnd),                 % get selection end position
   (SelStart = SelEnd,                        % account for a null selection
     Posn2 = SelEnd;                          % if null then end position is correct
     Posn2 is SelEnd - 1).                    % ...else adjust for the simulation
```