FORMAL SPECIFICATION

OF A

DISPLAY EDITOR

BERNARD SUFRIN

Technical Monograph PRG-21
June 1981

Oxford University Computing Laboratory
Programming Research Group
45 Banbury Road
Oxford OX2 6PE

Abstract

We present a formalisation of the design of a Display Editor The formalisation is rigorous enough to serve as a touchstone for the correctness of implementations of the editor and to permit various desirable properties of the design to be proven.

The formalisation is expressed in (slightly embellished) conventional mathematical notation, the specialised features of which are explained in the text.

In a companion paper [Sufrin 81b] we give criteria by which the correctness of implementations may be judged, and demonstrate how an implementation can be developed from the specification and shown to satisfy the correctness criteria

Oxford University Computing Laboratory
Programming Research Group
45 Banbury Road
Oxford OX2 6PE

Acknowledgements

I am deeply indebted to Jean-Raymond Abrial for introducing me to the art of specification, and for reawakening my interest in mathematics after it had been dormant for many years. Thanks also to Tony Hoare and Ib Sørensen for many fruitful discussions, to John Hughes for discovering a serious flaw in an earlier formalisation, and to Geraint Jones and Tim Clement for critically reading parts of the manuscript. It was the challenge of trying to formalise Richard Bornat's lovely but complicated screen-editor -- DED -- which began this enterprise.

The work is part of a programme of research into Software Engineering Methods supported by the United Kingdom Science and Engineering Research Council under grant GR/A/43124.

INTRODUCTION

In this monograph we present the formal specification of a display editor which has been in use at the Programming Research Group since December 1979 and is informally described in Appendix 1.

Our goal is to give a mathematical model which can serve both to communicate our ideas about editor design and to act as a touchstone for correctness of implementations. The design herein offers a reasonably comfortable human interface coupled with the possibility of implementation on fairly cheap hardware. We hope that our formalisation inspires other implementations. When the time comes to discuss a standard for editors then techniques such as those used here may provide an appropriate framework for concise and unambiguous definition.

The purpose of a formal system specification is to capture precisely and obviously the requirements of the system designer and his client -- independently of whether the structures and functions embodied in it are immediately implementable. In practice a formal specification will be used both in reasoning about properties of the system being specified and as a means of communication between designer, implementer, and users. It should be an adequate basis both for judgements about the correctness of implementations of the system to be made, and for conclusions about system behaviour to be drawn independently of (and preferably in advance of) implementation.

The goal of facilitating reasoning is not really compatible with the idea that the specification should itself be executable. Although it is very tempting when making constructive specifications to consider them simply as "very high-level programs" such temptation should be resisted -- the purpose of the specification is to express relationships (between system components) rather than the algorithms which maintain them. The design of algorithms has its place further along in the system development process. an "algorithmic" cast of mind at the specification stage can make the task of formalisation much more difficult. To summarise: neither the "realism" of the data abstractions employed nor its "runability" should be a matter of undue concern during the formalisation of a specification.

For this reason we feel free to define functions *implicitly* -- that is by giving pre- and post- conditions -- rather than *explicitly* -- by lambda-abstraction. The effect is to clarify our explanation and facilitate reasoning (albeit at the risk of specifying something which is not a function). The following example may help to clarify the difference: one can define subtraction either implicitly by:

$$-: \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$$

$$(\forall\ n1, n2: \mathbb{N} \mid n1 > n2)\ ((n1 - n2) + n2 = n1)$$

or explicitly by:

$$-: \mathbf{N} \times \mathbf{N} \rightarrow\!\!\!\!\!\!+ \; \mathbf{N}$$

$$- = (\lambda\ n1,\ n2 \mid n1 > n2)\ (pred^{n2}\ (n1))\ \textbf{where}\ pred = suc^{-1}$$

In the first case we have specified the relationship we wish to hold between the arguments and results of $-$. In the second we have constructed a function which can be proved to satisfy that relationship. It is often easier to do the former than it is to do the latter; indeed it is sometimes a simpler way of indicating the constraints within which some freedom of choice may be exercised by an implementation.

The Structure of this Specification

Our specification is in three main sections: first we present the specification of a document editing subsystem; next, we specify a document display subsystem; finally we give the desired relationship between these two independently specified components.

The structure itself signifies a major design decision. It is all too often the case that software designs reflect too closely the precise characteristics of hardware on which they will initially be implemented. We believe that this adversely effects the designer's ability to come up with a really simple and effective human interface -- not to mention a simple explanation of his or her design. A display editor should not be designed simply or even *mainly* around what appears on the display; the temptation to adapt the design to specific properties of the display device are then too great.

In our design, therefore, editor commands are explained solely as transformations on documents; the role of the display subsystem can be summarised as *keeping that part of the document which surrounds the cursor in view whilst minimising major screen changes*. We believe (with some empirical justification) that this makes it possible to implement the design on a wide variety of display devices.

An important design principle is reflected in the fact that we have not cluttered our definition with hosts of "features". Although we are quite happy for implementers to customise our specification to their own taste, we think that the temptation to encumber an editor with all the generality of a general-purpose string-processing *language* should be resisted. The payoff in this case is that the prose description of the editor is a mere four pages long and most of the editing tasks one ever needs to do can be done once these pages are understood

Another important principle is that the editor has no hidden modes! The interpretation of every key is fixed, rather than depending on some non-manifest aspect of the history of the edit session so far The effect of this is that prediction of the effect of a keystroke is simplified, and that many common editing actions eventually become *reflexes* rather than complicated keystroke sequences which must be consciously considered The effect of this is to enable the typist to concentrate on the composition of the document rather than the complexities of the editor interface.

Specification Style

A definition in denotational style would involve the invention of an *abstract syntax* of editor commands, the definition of a model for the editor states, and the definition of a *semantic function* to map commands into the state transforming functions which define their effects. The extremely simple structure of the editor command "language" does not warrant the invention of an abstract syntax, however, and we have chosen instead to stay in the world of semantics -- defining a state model and some state transformations.

It should be understood that most of the "commands" of the editor are intended to be invoked by *single keystrokes*. A "syntax" for commands is simply a picture of a keyboard; the corresponding semantic function just connects each key on the keyboard to one of the state to state functions.

Rather than confronting the reader with a great deal of detail all at once we have chosen to present the specification of the editing subsystem in stages. At each stage we present a mathematical model powerful enough to capture the design decisions we wish to illustrate.

Section 1: Editing Documents

1.1: A Simple Editor Model

The first editor to be specified has commands which permit insertion, deletion, and motion of just one character at a time. Every command takes effect at, and may change the position of, *the current position* in the document -- which for historical reasons will be called the *cursor*.

The essential characteristics of the state of such an editor are the *content* of the document being edited, and the *position* of the cursor in the document. We consider the cursor to be *between* characters rather than *at* a character, so these two characteristics can be captured by a pair of sequences of characters -- henceforth called a DOC. One sequence corresponds to that part of the document which *precedes* the cursor, the other to that which *follows* the cursor. The set of characters which may appear in documents will henceforth be denoted CH.

```
DOC ____________________________________________
|       seq[CH] × seq[CH]
|_________________________________________________
```

The following are the primitive DOC-transforming functions which will be used to specify the effects of editor commands.

```
_____________________________________________________
    del,
    move:       DOC ↛ DOC
    ins:        CH → (DOC → DOC)
    content:    DOC → seq[CH]
    ---------------------------------------------
    move = (λ l, r | l ≠ ⟨⟩)(head(l), ⟨last(l)⟩ * r)

    del  = (λ l, r | l ≠ ⟨⟩)(head(l), r)

    ins  = (λ ch)
               (λ l, r)(l * ⟨ch⟩, r)

    content = (λ l, r)(l * r)
_____________________________________________________
```

The functions head and last are partial -- their domains being the non-null sequences (head maps a sequence to the sequence consisting of all but its last element). It is evident from their definitions that move and del are also partial, and we record this fact in their signatures. Their exact domains are also recorded as part of the λ-expressions which define them:

Examples

```
((C U R R E N T), (sp P O S I T I O N))
                    move
((C U R R E N), (T sp P O S I T I O N))
                    del
((C U R R E), (T sp P O S I T I O N))
                    del
((C U R R), (T sp P O S I T I O N))
                  insert(Y)
((C U R R Y), (T sp P O S I T I O N))
```

The functions so far defined are not sufficient to give the semantics for even the simplest editor since there is nothing corresponding to rightward motion. Rather than remedy this *ad-hoc*, we introduce a *method* for defining direction, which we will use throughout the monograph

```
mirror: DOC → DOC
mirror = (λ l, r)(reverse(r), reverse(l))
```

Example `mirror` applied to:

((F O O), (B A Z))

gives:

((Z A B), (O O F))

following this by a `move` we get:

((Z A), (B O O F))

then by another `mirror` we get:

((F O O B), (A Z))

in other words a rightward move!

It is easy to see that rightward delete and rightward insert can be defined similarly. We therefore define the following "directional" combinators.

```
right,
left:    (DOC↛DOC) → (DOC↛DOC)

right = (λ f)(mirror ∘ f ∘ mirror)

left  = (λ f)(f)
```

The function right maps any DOC function to its "rightward" counterpart, whilst left is simply a mnemonic renaming of the identity on DOC to DOC functions. We can now define the semantics of a family of simple editors whose commands correspond to the functions.

```
left(move)      left(del)      left(ins(c))   (for all c:CH)
right(move)     right(del)     right(ins(c))  (for all c:CH)
```

and have the following desirable properties:

1. An insertion followed by a deletion has no net effect on the document.

2. A move in one direction followed by a move in the opposite direction has no net effect.

3. A deletion in one direction can be achieved by a move in that direction followed by a delete in the opposite direction.

4. Motion has no net effect on the content of a document.

Proving the theorems which formalise these properties is relatively easy and is left as an exercise for the reader.

DOC PROPERTIES

```
⊢ (∀ c: CH)
        (left(del) ∘ left(ins(c)) = id(DOC) ∧
         right(del) ∘ right(ins(c)) = id(DOC))

⊢ left(move) ∘ right(move) = id(dom(right(move))) ∧
  right(move) ∘ left(move) = id(dom(left(move)))

⊢ left(del) ∘ right(move) = right(del) ∧
  right(del) ∘ left(move) = left(del)

⊢ content ∘ left(move) = content ∧
  content ∘ right(move) = content
```

Hint. use the following generic properties of sequences:

```
                              X
SEQ PROPERTIES ____________________________________________
    ⊢ (∀ s, s1, s2:  seq[X]; x: X)
        head(s)^tail(reverse(s))
        reverse(reverse(s)) = s
        reverse(s1 * s2) = reverse(s2) * reverse(s1)
        head(s * ⟨x⟩) = s
        last(s * ⟨x⟩) = x
```

Quantifying over ''Actions'' and ''Directions''

When giving the semantics of editors we will often define higher-order functions whose arguments and results are variables quantified over the functions which we have just defined. For this reason we give names to two classes of function.

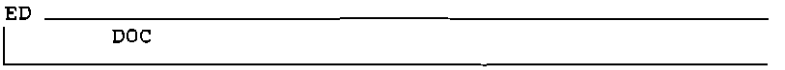
The ACTIONs are a subset of the DOCument-transforming functions, and the DIRECTIONs are a subset of the (higher order) functions on the DOCument-transforming functions.

```
ACTION:      P(DOC↔DOC)
DIRECTION    P( (DOC↔DOC) → (DOC↔DOC) )
-----------
ACTION =     {del, move}
DIRECTION =  {left, right}
```

1.2: A Simple Editor

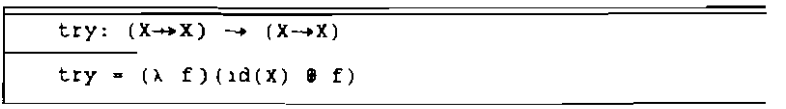
The DOC model and the functions thereon say a great deal of what needs to be said of *any* simple editor. In order to define the interface to one particular editor with greater precision we need to present a more detailed specification of the available commands.

As an example of our general approach we present here a simple editor. The effect of each of its commands will be modelled by a state-to-state function. Editor states -- henceforth denoted ED -- are modelled here by a single DOC.



We face a small problem in using the functions defined above to specify the effect of commands on editor states: despite the fact that we wish the effect of every command to be completely defined, the functions on DOC are not all total. For example, `right(move)` is only defined for a DOC which has text following the cursor; despite this we wish to specify the effect of an attempt to move the cursor rightwards beyond the end of the document as "no change to the content of the document nor the position of the cursor."

We can resolve the problem by defining a very general combinator, `try`, which maps a partial function, `f`, into a total function which agrees with `f` on its domain (a subset of the *generic* set `X`) and elsewhere agrees with the identity function on `X`.



This *generic* form of definition -- signified by the `X` above the double bar -- denotes a schema which can be instantiated for any *actual* set. Thus for a set `Y`

`try[Y]` denotes a function of type `(Y↛Y)→(Y→Y)`

Examples:

```
try[ED](move) (⟨e n⟩, ⟨d⟩) = (⟨e n d⟩, ⟨⟩)
```

but:

```
try[ED](move) (⟨e n d⟩, ⟨⟩) = (⟨e n d⟩, ⟨⟩)
```

It is customary to omit the *generic argument* ([ED] above) when invoking generically-defined functions, since their generic type is usually evident from context.

In the simple specification which appears below, the (state-to-state functions which model the effects of) commands come in two families: the FUNCTION commands (each of which is particularised by a DIRECTION and an ACTION) and the INSERT commands (particularised by a specific CHaracter).

```
INSERT:   CH → (ED → ED)
FUNCTION: (DIRECTION×ACTION) → (ED → ED)
```

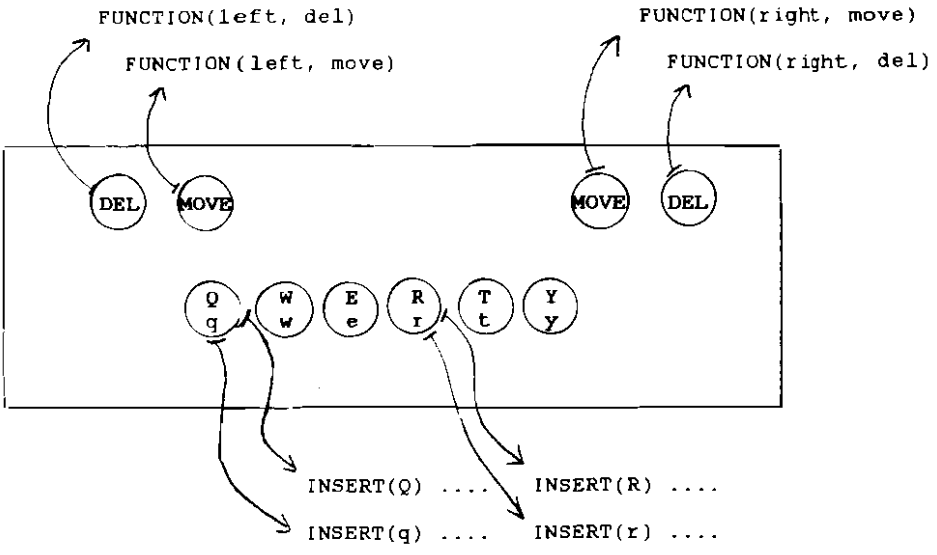
```
(∀ a: ACTION; d: DIRECTION; c: CH)
    INSERT(c) = ins(c)
    FUNCTION(d, a) = try(d(a))
```

We summarise the available commands by defining cmd as a subset of the ED-transforming functions, *ie* the union of the ranges of the functions INSERT and FUNCTION. The main design decision recorded here in addition to the properties which these commands "inherit" from the DOC model is the exclusion of rightward insertion from the command repertoire.

```
cmd:   P(ED → ED)
```

```
cmd = ran(INSERT) ∪ ran(FUNCTION)
```

One of the tasks of the designer of an *implementation* of this aditor will be to provide a mapping from *keys* on the chosen keyboard to each of the cmds specified above. We know of no ergonomic principles to guide the design of such a mapping although it is clear that the "visible" characters on the keyboard should be mapped to INSERT commands, and that the four FUNCTION commands should be mapped to special "function keys" if possible.



Keyboard Design for the Simple Editor

1.3: Enriching the Simple Model

Defining Significant Places in Documents

The next level of specification includes commands which act on larger units -- words, lines, and the whole document. In order to be able to specify such commands we need to enrich our simple model by formalising the idea of *word*, *line* and *document* boundaries.

We introduce a (constant) *newline* character and define the set -- line -- of documents whose cursor is positioned at the beginning of a line.

```
nl: CH
```

```
line:        P(DOC)
line = { l, r | l = <> v last(l) = nl }
```

Example:

$$(\langle l\ a\ s\ t\ \mathbf{nl}\ \rangle,\ \langle n\ e\ x\ t\rangle) \in line$$

but:

$$(\langle l\ a\ s\ t\ \rangle,\ \langle \mathbf{nl}\ n\ e\ x\ t\rangle) \notin line$$

The image of this subset of the documents through mirror is written:

$$\mathbf{mirror}(\ line\ )$$

(bold parentheses denote "image") and is equivalent to the set:

$$\{\ l, r \mid r=\langle\rangle \vee first(r)=nl\ \}$$

Example

$$(\langle l\ a\ s\ t\ \rangle,\ \langle \mathbf{nl}\ n\ e\ x\ t\rangle) \in \mathbf{mirror}(\ line\ )$$

In other words, the mirror image characterises the documents whose cursors are positioned at the end of a line!

By introducing a different constant -- *space* -- we can define the set of documents whose cursor is positioned at the left hand end of a word. It is convenient to have this set include the line boundaries.

```
sp: CH
---
sp ≠ nl
```

```
word:        P(DOC)
---
word = { l, r | l=⟨⟩ ∨ last(l)∈{sp, nl} ∧
                r=⟨⟩ ∨ first(r)∉{sp, nl}} ∪ line
```

Examples:

```
(⟨p u r p l e sp sp⟩, ⟨p r o s e⟩) ∈ word
(⟨f i n a l nl⟩, ⟨d r a f t⟩) ∈ word
```

but:

```
(⟨p a i n f u l⟩, ⟨sp p r o b l e m⟩) ∉ word
```

The mirror image of this set of documents is:

```
mirror( word ) =
            mirror( line ) ∪
            { l, r | l=⟨⟩ ∨ last(l)∉{sp, nl} ∧
                     r=⟨⟩ ∨ first(r)∈{sp, nl}}
```

Example:

```
(⟨p a i n f u l⟩, ⟨sp p r o b l e m⟩) ∈ mirror( word )
```

Thus the mirror image of word adequately describes the set of documents whose cursors are at the end of a word

Another significant set of documents are those whose cursors are at the beginning of the document. Finally there are those whose cursors are positioned at a character boundary -- *viz* the entire set of documents!

```
document:    P(DOC)
character:   P(DOC)
---
document = { l, r | l=⟨⟩ }
character = DOC
```

It will be useful to give names to the functions `mirror` and `id(CUT)` which more closely reflect the role they can play in specifying which "side" of a word, line or document is meant.

```
beginning,
ending:        DOC→DOC
-----------------------
beginning = mirror
ending    = id(DOC)
```

The following sets correspond to documents whose cursors are positioned at "significant" boundaries:

```
beginning( character )        ending( character )
beginning( word )             ending( word )
beginning( line )             ending( line )
beginning( document )         ending( document )
```

with the obvious property that

```
beginning( character ) = ending( character ) = character
```

Since we later want to quantify over the "sides" and "places" we define the two sets SIDE and PLACE.

```
SIDE:  P(ED→ED)
PLACE: P(P(DOC))
----------------------------------------
SIDE  = {beginning, ending}
PLACE = {character, word, line, document}
```

Motion and Deletion to Significant Places

In order to specify motion or deletion "to the next (previous) ...", we first define functions which map documents into their *distance* (ie number of moves) to the nearest "..." in a given direction *if there is such a place*. Notice that we have not given an *algorithm* which *discovers* the distances, simply specified what they are.

```
dist: (DIRECTION×P(DOC)) → (DOC ↛ N)

dist =
  (λ dir, place)
    (λ doc | distances ≠ {} ) (min (distances))
    where distances =
          {d:N | d>0 ∧ dir(move)^d (doc) ∈ place}
```

Example:

dist(right, ending(word))(⟨h e r⟩, ⟨sp h a n d sp i s⟩) = 5

but

(⟨h e r⟩, ⟨sp h a n d sp i s⟩) ∉ dom(dist(right, ending(line)))

Moving or deleting to a given place in a given direction is specified simply as an iteration of the action. The effect is unspecified for documents in which the place is unreachable.

```
to: ((ACTION×DIRECTION)×P(DOC)) → (DOC↛↛DOC)

to = (λ action, dir, place)
       (λ doc | doc ∈ dom(dist(dir, place))
          (dir(action)^n (doc))
       where n = dist(dir, place)(doc)
```

Example:

(del, right) to ending(word)

maps

(⟨h e r⟩, ⟨sp h a n d sp i s⟩)

to

(⟨h e r⟩ ⟨sp i s⟩)

The functions:

```
(move, left)  to character
(move, right) to character
(del, left)   to character
(del, right)  to character
```

obviously correspond to `left(move)`, `right(move)`, `left(del)`, and `right(del)` of our original model.

The functions:

```
(move, left) to beginning(word)
(del,  left) to beginning(word)
(move, left) to beginning(line)
(del,  left) to beginning(line)
(move, left) to beginning(document)
(del,  left) to beginning(document)

(move, left) to ending(word)
(del,  left) to ending(word)
(move, left) to ending(line)
(del,  left) to ending(line)
(move, left) to ending(document)
(del,  left) to ending(document)
```

together with their rightward-acting counterparts are suitable for modelling commands whose properties are obvious generalisations of the desirable properties of the simple model.

1. Moving from a (character, word, line) boundary to the same kind of boundary then moving back in the opposite direction to the same kind of boundary, has no net effect.

2. A deletion can be achieved by motion in the appropriate direction followed by deletion in the opposite direction.

3. Motion has no net effect on the content of a document.

These properties are formalised by the following theorem, whose proof is once more left as an exercise for the reader.

PLACE THEOREM

```
⊢ (∀ place: P(DOC))
      lmove  ∘ rmove = id(dom(rmove)) ∧
      rmove ∘ lmove  = id(dom(lmove)) ∧

      rdel = ldel  ∘ rmove ∧
      ldel = rdel ∘ lmove  ∧

      content ∘ try(lmove) = content ∧
      content ∘ try(rmove) = content

  where rmove = ((move, right) to place)
  and   lmove  = ((move, left)  to place)
  and   ldel   = ((del, left) to place)
  and   rdel   = ((del, right) to place)
```

Hint: use the following lemmas.

DIRECTION LEMMAS

```
⊢ mirror ∘ mirror = id(DOC)

⊢ right ∘ right = id(DOC↔DOC)

⊢ (∀ n: ℕ) right(move)^n = right( move^n )
```

Correcting Erroneous Deletions

We wish to define an editor in which fairly large amounts of text can be deleted at a single keystroke. Since it is all too easy to mistakenly hit a key we include a "recall" command, which undoes the most recent delete command.

In order to specify the effect of such a command we enrich the DOC model further by defining three functions on DOCs which are analogous to concatenation and sequence difference. We also define two relations analogous to sequence prefix and suffix

Notice that the difference functions are specified implicitly (ie by pre- and post-conditions) rather than explicitly.

```
**:          DOC×DOC → DOC
//:          DOC×DOC ↛ DOC
\\:          DOC×DOC ↛ DOC
infixes:     DOC ↔ DOC
outfixes:    DOC ↔ DOC
```

```
** = (λ (l, r), (l', r'))(l * l', r' * r)

(∀ d, d': DOC)
     (d infixes d') ⇔ (∃ d'': DOC | d'' ** d = d')
     (d outfixes d') ⇔ (∃ d'': DOC | d ** d'' = d')

(∀ d1, d2: DOC | d2 outfixes d1)
                d2 ** (d1 // d2) = d1

(∀ d1, d2: DOC | d2 infixes d1)
                (d1 \\ d2) ** d2 = d1
```

Examples.

```
(⟨F O⟩, ⟨A Z⟩) ** (⟨O B⟩, ⟨⟩) = (⟨F O O B⟩, ⟨A Z⟩)
(⟨F O⟩, ⟨A Z⟩) outfixes (⟨F O O B⟩, ⟨A Z⟩)
(⟨O B⟩, ⟨⟩) infixes (⟨F O O B⟩, ⟨A Z⟩)
(⟨F O O B⟩, ⟨A Z⟩) // (⟨F O⟩, ⟨A Z⟩) = (⟨O B⟩, ⟨⟩)
(⟨F O O B⟩, ⟨A Z⟩) \\ (⟨O B⟩, ⟨⟩) = (⟨F O⟩, ⟨A Z⟩)
```

It is evident that every deletion maps a document to one of its outfixes, and that consequently the outer-difference between a DOCument and a deletion applied to it is well-defined. More formally:

```
⊢ (∀ d: DOC; dir: DIRECTION; n: ℕ | d∈dom( dir(del)^n ))
                (dir(del)^n (d) outfixes d)
```

1.4: The Basic Document Editor

We now define a more powerful editor which supports character word and line motion and deletion, together with a limited form of recovery from erroneous deletions.

The editor state has two components, namely the text being edited and the "last deletion".

```
ED
   text:     DOC
   deleted:  DOC
```

Editor commands are, as before, modelled by ED to ED functions -- of which there are now three families, namely the FUNCTION commands, the INSERTion commands, and the RECALL command. Notice that neither insertion nor motion commands affect the last deleted text. This means that a rudimentary form of "cut and paste" can be performed by deleting, moving and recalling. A more general form of cut and paste is specified in Appendix 2.

```
FUNCTION:  (DIRECTION×ACTION×SIDE×PLACE) → (ED → ED)
INSERT:    CH → (ED → ED)
RECALL:    ED → ED

(∀ d: DIRECTION; a: ACTION; s: SIDE; p: PLACE; c: CH)

 INSERT(c) =
        (λED)
          (μED')
            text'=ins(c)(text);
            deleted'=deleted

 FUNCTION(d, a, s, p) =
        (λED)
          (μED')
            text'=try( (a,d) to s( p ) )(text);
            a=del=>
              deleted'=text // text';
            a=move=>
              deleted'=deleted;

 RECALL =
        (λED)
          (μED')
            text'=text ** deleted;
            deleted'=deleted
```

Notational Interlude:

The expression

```
(λED) ...
```

denotes exactly the same function as.

```
(λ text: DOC; deleted: DOC) ...
```

In which the text of the ED schema is substituted for the occurence of ED The names of the components of ED are *bound* within "..." by the quantifier λ.

The expression `(μED') ... list of predicates ...`

denotes an element of ED for which all of the given predicates hold (μ is pronounced "make-an"). The names of the components of ED', *ie* `deleted'` and `text'` are *bound* by the quantifier μ, just as the names `deleted` and `text` were bound by the quantifier λ. The "dashing" permits components of the arguments and results of λ-μ functions to be distinguished.

Writing down a μ-expression does not in itself guarantee the existence of an element with the required properties; this must (in general) be *proven independently* by showing that the predicates are mutually consistent. In the above case this is trivially evident.

(For a more comprehensive explanation of the "schema" notation and its relation to mathematical quantifiers see [Sufrin81a]).

Properties of the Recall Command

It is evident from the above definition that the RECALL command corrects the effect of any erroneous deletion on the textual component of the state. More formally:

```
⊢ (∀ e: ED; dir: DIRECTION; side: SIDE; place: PLACE)
    (RECALL( deletion (e) )).text = e.text
    where deletion = FUNCTION(dir, del, side, place)
```

As before we summarise the commands which should be made available to the user. There are actually fewer *useful* FUNCTIONs than there might seem to be at first sight. For example: the end of a document can never be found to the left of the cursor, the beginning of a document is never to the right of the cursor and the beginning and ending of a character are identical

The function delete left to ending of line and the function delete right to beginning of line are needed so infrequently that they can be omitted from the keyboard and simulated with a couple of other keystrokes when necessary.

We found in practice that the availability of both the beginning(word) and the ending(word) functions complicates the keyboard designer's task and gives the typist too many ways of performing word-related tasks. The ending(word) functions are best left out.

Our decision to omit these commands is recorded in in the following definition of the set of available commands.

```
cmd: P(ED → ED)

cmd = ran(INSERT) ∪ ran(FUNCTION) ∪ {RECALL} - excluded

where excluded =
    FUNCTION( DIRECTION×ACTION×{ending}×{word} ) ∪
    FUNCTION( {right}×ACTION×{beginning}×{document} ) ∪
    FUNCTION( {left}×ACTION×{ending}×{document} ) ∪
    {FUNCTION(left, delete, ending, line)} ∪
    {FUNCTION(right, delete, beginning, line)}
```

Suggested Keyboard Design (Partial)

When shifted the function keys marked CHAR, WORD and LINE should map to deletions, otherwise they should map to motions.

1.5: The Complete Document Editor

The commands defined in the previous section form the basic repertoire of the editor. In this section we show how they can be used to form patterns and replacement texts for the searching and substitution commands.

We begin to specify these commands by ignoring the problem of how the typist supplies the patterns and replacements to the editor. The find functions move the cursor in the appropriate direction to the nearest place in the document which matches a given pattern (if such a match occurs). The replace functions remove the matching text from a document positioned at an instance of a pattern, and insert the replacement in its place.

```
find:     (DOC×DIRECTION) → (ED↔ED)
replace:  (DOC×DOC) → (ED↔ED)

find = (λ pattern, dir)
          (λED | text∈dom( (move,dir) to match) )
             (μED')
               text'=( (move,dir) to match )(text)
               deleted'=deleted
          where match = { d: DOC | pattern infixee d }

replace = (λ pattern, repl)
             (λED | pattern infixes text)
                (μED')
                  text' = (text \\ pattern) ** repl
                  deleted' = pattern
```

A formal definition in a more traditional style might leave to the implementer the task of choosing a method for the typist to supply patterns for the find command and replacements for the replace command. For a highly interactive program the human interface needs to be specified more precisely, and so we specify below (first informally, and then formally) the details of a suitable method of constructing patterns and replacements.

Hitherto every character typed has been inserted into the current document. In order to specify patterns and replacements we need to be able to type text which does not have an immediate effect on the document. The QUOTE key signals the start of such a text -- which can be composed using all of the basic editing commands and without having any effect on the document itself. The FIND and REPLACE keys signal the completion of pattern (replacement) text, and have an appropriate effect. In the case of the FIND key the effect is to move the cursor (if possible) to the next instance of the pattern in the document. The REPLACE key changes the text of a document *which is already positioned at an instance of the FIND pattern* by substituting the replacement text for the FIND pattern.

Thus to change the next instance of FOO to BAZ, one types

QUOTE F O O **FIND QUOTE** B A Z **REPLACE**

Both the **FIND** and the **REPLACE** keys remember their last argument, so that (for example) to replace the next instance of FOO one simply types:

FIND REPLACE

whereas to delete the next instance one would have typed:

FIND QUOTE REPLACE

subsequent deletions being performed by:

FIND REPLACE

Later we will show that these definitions do not contradict our strictures about hidden modes: one of the properties specified of the complete editor is that QUOTEd text is always displayed.

We now render more formally the informal definition given above. As before we specify the effects of these commands as state to state functions

Editor states are denoted by the set EDITOR, whose `main`, `quoted`, and `mode` components reflect the fact that one is either editing the main document, or composing quoted text which will be used either as a pattern or a replacement. The `pattern` and `replacement` components reflect the fact that the FIND and REPLACE keys remember their last arguments.

```
EDITOR ____________________________________________
  main:        ED
  quoted:      ED
  pattern:     DOC
  repl:        DOC
  mode:        {MAINTEXT, QUOTEDTEXT}
```

The effect of each key depends on whether one is composing the QUOTEd or the **MAIN** text. Each key is therefore defined in terms of two *partial functions* whose domains correspond to the two different modes.

In what follows we have omitted predicates of the form (component'=component) simply in order to make the function definitions more compact.

```
BASIC:    cmd → (EDITOR → EDITOR)
FIND:     DIRECTION → (EDITOR → EDITOR)
QUOTE:    (EDITOR → EDITOR)
REPLACE:  (EDITOR → EDITOR)

(∀ c: cmd; dir: DIRECTION)
 BASIC(c)  =
          (λEDITOR | mode = MAINTEXT)
            (μEDITOR')
              main' = c(main)
         ⊕(λEDITOR | mode = QUOTEDTEXT)
            (μEDITOR)
              quoted' = c(quoted)                          1

 QUOTE  =
          (λEDITOR | mode = MAINTEXT)
            (μEDITOR')
              quoted' = EmptyED;
              mode' = QUOTEDTEXT
         ⊕(λEDITOR | mode = QUOTEDTEXT)                    2
            (μEDITOR')
              quoted' = EmptyED;
              mode' = MAINTEXT

 FIND(dir)  =
          (λEDITOR | mode = MAINTEXT)
            (μEDITOR')
              main' = (try(find(pattern, dir)))(main)      3
              quoted' = EmptyED
         ⊕(λEDITOR | mode = QUOTEDTEXT)
            (μEDITOR')
              main' = (try(find(pattern', dir)))(main)     4
              quoted' = EmptyED
              pattern' = (<>, content(quoted.text))        5
              mode' = MAINTEXT

 REPLACE  =
      (λEDITOR | mode = MAINTEXT)
        (μEDITOR')
          main' = (try(replace(pattern, repl)))(main)      6
          quoted' = EmptyED
     ⊕(λEDITOR | mode = QUOTEDTEXT)
        (μEDITOR')
          main' =  (try(replace(pattern, repl')))(main)    7
          quoted' = EmptyED
          repl' = quoted.text                              8
          mode' = MAINTEXT

where EmptyED = (μED)
                  text = (<>, <>)
                  deleted = (<>, <>)
```

To help explain our design decisions we have numbered several of the predicates; the corresponding annotations follow.

1. Any of the basic commands can be used to compose quoted text. Their effect on the quoted text is identical to their effect on the document.

2. This is actually a suitable place to incorporate commands which interface the editor to its environment. Although their formal definition is beyond the scope of this monograph, the commands for leaving the editor and for aborting an editor session are typed in our implementations as:

   QUOTE q **QUOTE**
   QUOTE a b o r t **QUOTE**

3. An unquoted FIND uses the pattern remembered from the last quoted FIND.

4. & 5. A quoted FIND uses the quoted text to form its pattern. FIND will always position the cursor so that it is at the *beginning* of the text matching the pattern. This is more useful in practice than positioning the cursor at the end of the match.

6. An unquoted REPLACE uses the pattern remembered from the last quoted FIND and the replacement from the last quoted REPLACE.

7. & 8. A quoted REPLACE uses the quoted text to form its replacement. The cursor position relative to the replacement text will be the same as its position within the quoted text.

Once more we can summarise the commands which are to be made available, observing that these include all the commands of the basic editor, to which we have added the two FIND commands, the replace command and the QUOTE command:

key: P(EDITOR → EDITOR)
key = ran(BASIC) ∪ ran(FIND) ∪ {REPLACE, QUOTE}

Section 2: Displaying Documents

A Simple Display Model

Editor commands were specified as transformations on the more-or-less one-dimensional DOC modal. In order to specify the way in which documents are displayed we need a model which reflects the fact that most(!) real displays are two-dimensional and bounded. The boundedness of real displays forces us to have some sort of policy for choosing which part of the document to display. In our view the best choice is to display that part of the document which surrounds the current position, emphasising the current position on the display by means of some distinctive symbol (such symbols are usually called *cursors*). This policy can be likened to looking at the document through a movable *window* which locates itself so as to keep the cursor in view. It has the advantage that when changing a document the typist always sees on the display an exact picture of the region of the document which has most recently changed, and can quickly discover whether or not the changes are the ones intended.

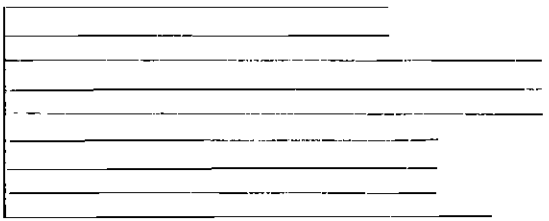
We begin to formalise the idea of a two-dimensional display by ignoring the fact that real ones are bounded. A LINE is a sequence of CHaracters which doesn't contain the newline character nl. An unbounded display and the position of its cursor can be completely characterised by four quantities -- the sequence of LINEs above the cursor, the sequence of LINEs below the cursor, and the sequences of CHaracters to the left and the right of the cursor (neither of which contain newline characters).

```
LINE ____________________________________________
| seq[CH-{nl}]
|_________________________________________________

DISP ____________________________________________
| above,
| below:      seq[LINE]
| left,
| right:      LINE
|_________________________________________________
```

above

left ⟨The cursor is between⟩ ⟨these two sequences⟩ **right**



below

Relating Unbounded Displays to Documents

We formalise the correspondence between an unbounded display and a document by developing a one-to-one relation -- `displays`. First we define the function `flatten` -- which maps a (nonempty) sequence of lines to a sequence of characters which has newline characters separating the text of the original lines. (Notice that it is specified by a postcondition reminiscent of a *recursive definition*. Theoretical justification for the existence of functions satisfying such postconditions is given more fully in [Sufrin81a]).

```
flatten:    seq1[LINE] → seq[CH]

(∀ ln: LINE; s: seq1[LINE])
 flatten(⟨ln⟩) = ln    ∧
 flatten(⟨ln⟩ * s) = ln * ⟨nl⟩ * flatten(s)
```

Example:

```
flatten(⟨⟨l L I N E⟩⟩) = ⟨l L I N E⟩
flatten(⟨⟨L I N E l⟩ ⟨L I N E 2⟩⟩) = ⟨L I N E l nl L I N E 2⟩
```

An unbounded display corresponds (through the `displays` relation) to a DOCument under the following conditions.

1. Flattening the lines above and to the `left` of the display cursor gives the sequence of characters to the left of the document cursor, and
2. Flattening the lines to the `right` and below the display cursor gives the sequence of characters to the right of the document cursor.

```
displays: DISP ↔ DOC

(∀ disp: DISP; (l,r): DOC)
 (disp displays (l,r)) ⟺
       (flatten(disp.above * ⟨disp.left⟩) = l) ∧
       (flatten(⟨disp.right⟩ * disp.below) = r)
```

For example:

```
⟨ ⟨line1⟩
  ⟨line2⟩
  ⟨line3⟩ ⟩
  ⟨left⟩ ⟨right⟩
⟨ ⟨line5⟩
  ⟨line6⟩ ⟩

displays (⟨line1nlline2nlline3nlleft⟩, ⟨rightnlline5nlline6⟩)
```

It is easy to prove that `flatten` is a bijection, *ie* that it maps different (nonempty) sequences of lines into different sequences of characters and *vice-versa*. An easily-proven consequence of this is that the `displays` relation is also a bijection, *ie* that every unbounded display corresponds to a unique DOCument, and *vice-versa*. This is hardly surprising: our initial one-dimensional formalisation of documents would have been rather implausible if not for our intuition that such a one-one correspondence existed. This property will again prove useful when we are considering an implementation of the editor.

FLATTEN LEMMA

```
⊢(∀ s1, s2: seq1[LINE])
    (flatten(s1)=flatten(s2) ⟺ s1=s2)
```

DISPLAY THEOREM

```
⊢(∀ d1, d2, d: DISP; doc1, doc2, doc: DOC)
  ((d1 dieplays doc) ∧ (d2 dieplays doc)) ⟺ d1=d2 ∧
  ((d displaye doc1) ∧ (d displays doc2)) ⟺ doc1=doc2
```

(The formal proofs are omitted here since they are of no intrinsic interest).

Displays and Windows

In order to formalise the idea of a *window* we observe that the content of a display can be mapped onto an irregular two-dimensional character matrix, and that its cursor position can be modelled by a pair of numbers. Since the DISP model puts the cursor between characters, each line has one more cursor position than it has characters, and so the column index of the cursor corresponds to the position of the character to its left.

```
cursor: DISP  →  (N × N)
matrix: DISP  →  ( (N × N) → CH )

cursor =  (λDISP)(1+#above, #left)
matrix  = (λDISP)
           (λr, c | r∈(1..#lines) ∧ c∈(1..#(lines(r))))
            (lines(r)(c))
           where lines = above * ⟨left * right⟩ * below
```

Notice that of the three concatenation operators defining the sequence `lines`, the outer two concatenate sequences of LINEs whereas the inner one concatenates sequences of CHaracters

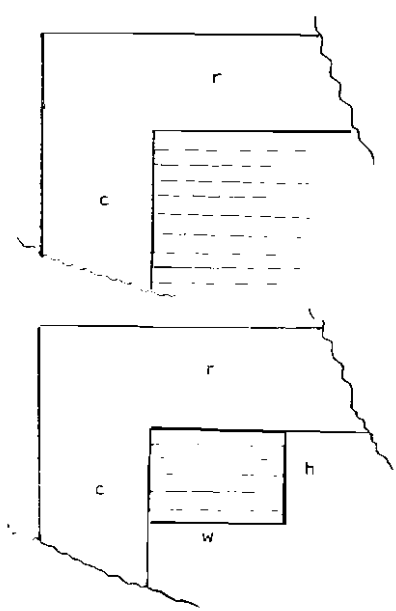
A fundamental property of the cursor of a display is that it corresponds to a character on the matrix of the display unless it is at the left-hand end of a line. Consequently displays cannot be constructed from *arbitrary* character matrices and cursor positions.

```
CURSOR LEMMA
⊢ (∀ d: DISP; r, c: N | (r, c)=cursor(d))
      ((r, c)∈dom(matrix(d)) ⇔ (c≠0))
```

The functions `project` and `region` provide the additional tools we need to formalise the idea of a window. Composition with a `project`ion function corresponds to moving the origin of a matrix. Restricting its domain by a `region` corresponds to limiting its "area"

```
project: (N × N) → ( (N × N) → (N × N) )
region:  (N × N) → P(N × N)

project = (λ r, c)(λ i, j)(r+i, c+j)
region  = (λ h, w)((1..h) × (1..w))
```



d project(r, c)

f region(h, w)

For given constant screen dimensions (height and width) the function **window** *maps* a window offset (row and column) into a function on DISPlays which has the following properties:

1. The character *matrix* of its result fits into the required dimensions and agrees exactly with the (row, column) projection of the *matrix* of its argument.

2. The cursor position of its *result* is *the same projection* of the cursor position of its argument

The function is partial because it is not possible to project the cursor correctly for all choices of window offset A correct projection leaves the cursor either at the left hand end of a windowed line or corresponding to some character on the windowed display matrix

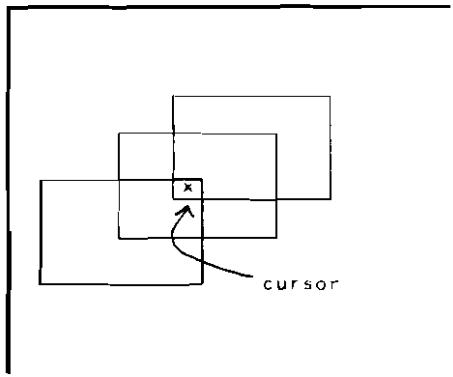
```
height:   N
width:    N
window:   (N × N) → (DISP ↛ DISP)
```

```
(∀ r, c: N; d, d': DISP)

  d'=window(r, c)(d) ⇔ (matrix(d')=wm ∧ cursor(d')=wc)

  where wm = (matrix(d)∘project(r, c))↑screenarea
  and   wc = project(r, c)(cursor(d))
  and   screenarea = region(height, width)
```

It is evident, however, that for a given document there are several possible choices of window offset which *do* project the cursor correctly, for example:



It is easy to derive an expression which gives the precise relation between window origin and correct projection A window whose origin is (r c) can correctly project a display whose cursor falls within the appropriately projected screen area

```
WINDOW THEOREM
⊢ (∀ r, c: N)
    dom(window(r, c)) = { d: DISP | cursor(d)∈screen }
    where screen = project(r, c)( region(height, width) )
```

Section 3: Displaying the Edited Document

The Complete Editor State

In this section we formalise the hitherto informally stated requirement that the screen of the display should be a window onto the (unbounded display corresponding to the) document being edited.

The function `display` maps the state of the document-editing module to a "virtual" display which the display module must window. Specification of just what should be displayed is slightly complicated by the need to see quoted text whilst it is being prepared and to distinguish it from the body of the main document. The method we have suggested here is to "embed" quoted text in the display of the main document -- separating it from the document text by (implementation dependent) sequences which play the role of quotation marks.

```
quote:   seq[CH]
unquote: seq[CH]
```

```
display: EDITOR → DISP

display = (λ EDITOR)(virtual)

where mode=QUOTEDTEXT ⇒
          virtual displays document**quotes**quotation

and   mode=MAINTEXT ⇒
          virtual displays document

and   document  = main.text
and   quotation = quoted.text
and   quotes = (quote, unquote)
```

The state has a component -- `editor` -- which models the state of editing module, and components -- `screen`, `row`, and `col` -- which model the state of the display module. The single invariant on the state indicates the desired relationship between the two modules.

```
DISPLAYEDITOR
   editor:  EDITOR
   screen:  DISP
   row:     N
   col:     N

      screen = (window(row, col))(display(editor))
```

Specifying a Windowing Policy

The task of the editing module has already been specified (at least in the sense that we have given formalisations of the effects of every key as functions on its state). The task of the display module after each keystroke is to derive a *window offset* and the content of the screen from the editing module in such a way as to maintain the given invariant. It is evident from the considerations of the previous section that in general there will be some freedom to choose the window offset, and that a *policy* will therefore be necessary.

We suggest an *incremental* windowing policy which tries to keep the window offset constant for as long as it can. When the cursor moves to a point where it no longer appears on the windowed portion of the display another displacement is selected. The advantages of this policy are principally.

1. When inserting material the typist does not get distracted by the task of finding the cursor after every keystroke, since it behaves "like pen on paper."

2. Most keystrokes result in relatively small changes to the content of the screen, and these take place at or near the cursor. This facilitates implementation on slowish dumb terminals.

We formalise a windowing policy function below by means of a postcondition. Given the current window offset, and a new virtual display the policy function must map them to an offset which permits the cursor to be shown on the screen.

Notice that the postcondition *does not uniquely define the policy* but gives an overall requirement for it. At this level of specification we are not too much concerned with the exact details of the policy. **any** function satisfying the requirements outlined below will suffice. This freedom will elsewhere allow a proof of correctness of a particular display strategy to be made independently of the details of the windowing policy.

```
policy:  (N × N) → (DISP → (N × N))

(∀ r, c:  N; d: DISP;
   r', c': N | (r', c') = policy(r, c)(d))

     d ∈ dom(window(r, c)) ⇒ (r', c')=(r, c) ∧
     d ∉ dom(window(r, c)) ⇒ d ∈ dom(window(r', c'))
```

We can now summarise the effect of a single keystroke on the state of the display editor.

```
effect: key → (DISPLAYEDITOR → DISPLAYEDITOR)

effect = (λ k)
           (λDISPLAYEDITOR)
            (μDISPLAYEDITOR')
               editor' = k(editor)
               (row', col') = policy(row, col)(virtual)
               screen' = window(row', col')(virtual)
             where virtual = display(editor')
```

The definition of `effect` concludes our specification. Proof that this function maintains the editor state invariant is extremely simple -- once again we leave it as an exercise for the reader.

4: Summary

The specification given herein is a revised and much-simplified version of the one from which our first implementation was built [Sufrin80a]. The simplification reflects our own deeper understanding of the techniques of formal specification. However in our treatment of the cut and paste and the automatic indentation facilities [Appendix 2] there seems to be some conflict between real-life useability and simplicity of specification. Both the facilities are extremely useful in the composition of text; their specifications, however, remain somewhat inelegant: indeed one can almost hear the whine of the machinery!

The fact that the proofs of many of the useful properties of the editor are easy enough to have been left as exercises reflects to some degree the amazing simplicity of our document and display models. Discovering such simple abstractions is a tough task, however, it seems only to come with a lot of experience and a very large waste-paper basket!

Again because of the simplicity of our models prospects for the design and proof of implementations are good. In a note to be published later [Sufrin81b] we have proven one class of implementation strategy correct. We use techniques similar to those described in [Jones] to show the correctness of our choice of data representation, and independently developed techniques to design our algorithms.

In this proof we rely heavily on the fact that the abstract display model developed in section 2 is also a natural basis for the formalisation of the properties of "smart" terminals. The obvious implementation strategy for this kind of display technology is for the editing module to give "hints" to the display module whenever the display needs to be changed. The hints indicate (where possible) the *incremental* changes to the screen which are necessary to maintain the screen-document relationship: when this is not sensible (for example after a CUT or a PASTE or when a search takes the current position off the screen) then the hint should indicate this, and the display module behave accordingly. This strategy seems to give good performance even on relatively dumb terminals -- the only time one consciously waits for the screen to reflect a change in the document is when the display module is forced to "pan" the window (*ie* move it horizontally), or to "tilt" it more than the height of the screen.

5: Conclusion

As we stated in our introduction, our goal was to give a mathematical model which serves both to communicate our ideas about editor design and to give an unambiguous definition against which the correctness of implementations might be proven.

In other engineering disciplines this sort of thing is a matter of course. Bridge-builders, architects, and aeroplane manufacturers all expect to have to reason formally about their artefacts before building them. They also have techniques by which to express their final designs unambiguously, and the craftspeople who transform these expressions into reality do so by means sound enough to ensure a faithful realisation.

Computer Scientists have over the last few years developed the intellectual tools necessary to permit the construction of large classes of program in just as sound a fashion. Case studies on the scale of this editor are still few and far between, however, and this has contributed to the reluctance of practising programmers to admit the possibility of reasoning formally about programs in advance of their construction or of verifying implementations against independently-specified blueprints.

Until their reluctance is overcome the phrase "Software Engineering" will remain no more than an empty piece of rhetoric. We hope that we have made a contribution to this goal.

Appendix 1: Informal Description of the Editor

The description which follows is extracted from a slightly revised version of the documentation of an implementation of the editor which has been in use since late 1979. So that our readers may judge for themselves the extent to which the formalisation can capture the behaviour of the implementation we have included the descriptions of some of the "custom" features of the implementation, *ie* those not described in the formal specification.

The editor permits the composition, alteration and examination of documents. The typist communicates with the editor through a keyboard which is equipped with a number of special function-keys; the document is shown on a screen on which the typist's current position is highlighted by means of a *cursor*. Once the document becomes too large to be seen on the screen, the editor ensures that the region surrounding the typist's current position is shown on the screen, for it is at or very close to the current position that all the typist's action have their effect.

Single-character insertion, deletion and motion

visible-character — The character is inserted in the document at the cursor; the cursor moves to the right; the remainder of the current line is pushed right to make room.

NEWLINE — Inserts a new line in the document at the cursor; any text to the right of the cursor becomes part of the new line.

In *auto-indent mode* (qv) enough spaces are put at the beginning of the line to place the first character of the new line below the first character of the previous line.

LEFT(DELETE — Rubs out the character to the left of the cursor. When the cursor is at the left-hand end of a line this joins the current line to the previous one.

RIGHT(DELETE — Rubs out the character at the cursor. When the cursor is at the right-hand end of a line this joins the line to the next one.

LEFT(MOVE — Moves the cursor leftwards one character position in the document. If the cursor was at the left-hand end of a line, then it will 'wrap around' to the right-hand end of the previous line.

RIGHT(MOVE — Moves the cursor rightwards one character position in the document. If the cursor was at the right-hand end of a line, then it will 'wrap around' to the left-hand end of the next line.

Multiple-character insertion, deletion and motion

A *WORD* begins where a space is followed by a character which is not a space.

LEFT(MOVE(WORD — Moves the cursor to the beginning of the previous *WORD*, or the start of the current line if that is nearer.

RIGHT(MOVE(WORD — Moves the cursor to the beginning of the next *WORD*or the end of the current line if that is nearer.

LEFT(MOVE(LINE — Moves the cursor to the beginning of the line, or the beginning of the previous line if it is already at the beginning of a line.

RIGHT(MOVE(LINE — Moves the cursor to the end of the line, or the end of the next line if it is already at the end of a line.

LEFT(DELETE(WORD — Deletes text between the cursor and the place to which *LEFT(MOVE(WORD* would move

RIGHT(DELETE(WORD — Deletes text between the cursor and the place to which *RIGHT(MOVE(WORD* would move.

LEFT(DELETE(LINE — Deletes text between the cursor and the place to which *LEFT(MOVE(LINE* would move

RIGHT(DELETE(LINE — Deletes text between the cursor and the place to which *RIGHT(MOVE(LINE* would move.

TAB — Inserts enough spaces in the line to put the cursor at the next tab position -- these are at eight column intervals.

Miscellaneous useful commands

MARK — Place the *mark* at the cursor.

CUT — Cut the text between the cursor and the mark out of the document.

PASTE — Insert the most-recently *CUT* text into the document at the cursor.

RECALL — Inserts the text deleted by the last word-delete, line-delete, or *REPLACE* back into the document.

MARGIN — Sets the right margin at the current column. When the right margin is set at a column other than the leftmost column, then whenever a character is typed to the right of the margin, the "word" of which it is a part will automatically be moved to the beginning of the next line.

NEXTPAGE — Display the next screenful of the document.
PREVPAGE — Display the previous screenful of the document.

Moving to specified places in the document

QUOTE — Inserts a new (empty) line at the cursor position; text to the right of the cursor appears below this line. The typist may now use any of the keys so far described to compose a *quotation* -- so called because the keys typed during its composition do not have an immediate effect on the document. The quotation is completed by typing one of the following keys: *FIND, UPFIND, REPLACE, QUOTE.* Its effect on the document depends on exactly which of these is used

If the quotation is completed by typing the *FIND* key, then the cursor is placed at the next place in the document which matches it. the *UPFIND* key moves UP the document to find the text. If there is no such place in the document then the cursor stays in the same place, and the bell rings. The quoted text be retained so there is no need to retype it in order to *FIND* the same text more than once.

If the quotation is completed by typing the *REPLACE* key, and the current FIND text matches the document at the cursor then it will be replaced by the quoted text. The quoted text will be retained so that there is no need to retype it in order to *REPLACE* with same text more than once.

If the quotation is completed by typing the *QUOTE* key, then the line is interpreted as a **special command**, and performed immediately

Special commands are:

t(op — Move the cursor to the top of the document.

b(ottom — Move the cursor to the end of the document.

q(uit — Leave the editor normally.

w(rite — Write the document to the output file but do not leave the editor (it's a good idea to periodically w(rite out a document which you are editing).

i*NAME* — Copy the document stored in the file called *NAME* into the current document below the current line.

o*NAME* — Make a new document from lines between the marked line and the current line. Store this document in the file called *NAME*.

mk — Place the mark at the cursor, then move the cursor to the place which was formerly marked.

i(ndent — Set auto-indent mode.

n(oindent — Clear auto-indent mode.

abort — Leave the editor, abandoning any work done this editing session

wd(find — Changes to *word match* mode *(qv)*.

lit(find — Changes to *literal match* mode *(qv)*.

FIND — Finds the last FIND text (downwards)

UPFIND — Finds the last FIND text (upwards).

REPLACE — If the text at the cursor matches the FIND text, then the text which was 'found' is replaced by the last *REPLACE* text.

Matching Criteria

In *literal match* mode, a *FIND* succeeds where the text at the cursor exactly matches the characters of the *FIND* text. In *word match* mode and when the *FIND* text consists entirely of letters or digits, a *FIND* succeeds only where the text at the cursor matches the *FIND* text and the characters immediately preceding and following the match are neither letters nor digits.

Appendix 2: Some Additional Features

In order to simplify the description given in this monograph a number of commands present in our implementation were omitted. It is beyond the scope of this monograph to specify some of these, amongst which are the input (output) of selected portions of the document from (to) the filing system and the automatic "filling" and/or justification of portions of the document.

Two other important omissions are formalised below:

Cut and Paste

This permits the typist to mark-up and move around pieces of text which extend over more than a single line. The output of selected portions of the document to the filing system is an obvious generalisation of the facility defined here.

Cut and paste can be added most conveniently to the specification of the basic editor. Extend the state defined there with an additional component which models the text removed by the last "cut"

```
ED ____________________________________________
   text:     DOC
   deleted:  DOC
   hold:     DOC
```

Three additional keys are provided. The MARK key places a special mark at the cursor, removing any other mark present in the document. The CUT key removes the text between the current cursor position and the mark from the document and places it in the hold buffer. The PASTE key inserts the content of this buffer into the document at the cursor.

```
mk:     CH
MARK:   ED → ED
CUT:    ED → ED
PASTE:  ED → ED
```

```
MARK = (λ ED)
         (μ ED')
            text' = removemark(text) ** mark;
            deleted' = deleted;
            hold' = hold

CUT = (λ ED)
        (μ ED')
          text' = removemark(cut(text))
          deleted' = deleted;
          hold' = removemark(text // text')

      where cut =
            try( ((del, right) to marked) ⊕
                 ((del, left)  to marked) )

      and   marked = {d: DOC | mark infixes d}
      and   mark = (⟨mk⟩, ⟨⟩)

PASTE = (λ ED)
          (μ ED')
            text' = text ** hold
            deleted' = deleted
            hold' = hold

where removemark = (λ l, r: seq[CH])(rmk(l), rmk(r))
and   rmk = (μ f: seq[CH] → seq[CH])
              f(⟨⟩) = ⟨⟩
              (∀ l: seq[CH]; c: CH | c≠mk)
                f(l * ⟨c⟩) = f(l) * ⟨c⟩
                f(l * ⟨mk⟩) = f(l)
```

In order for the effect of CUT to be easily predictable, it is evident that the domains of the rightward- and leftward- deleting functions from which cut is constructed should be disjoint -- this is true only for DOCuments which have at most one mk present. This can be ensured for all documents by defining the available commands so that the MARK key is the only one which inserts the mk character.

```
cmd: P(ED → ED)
```

```
cmd = ran(INSERT\{mk}) ∪ ran(FUNCTION) - excluded ∪
      {MARK, CUT, PASTE, RECALL}

where excluded = ... as defined in section 1.4 ...
```

The distance from the beginning of the line of the first "genuine" word boundary on the current line is called the "margin". When automatic indentation is enabled, pressing the NEWLINE key inserts a newline character followed by enough spaces to make the margin the same as on the previous line. This vastly simplifies the task of entering indented text.

```
margin:    DOC → ℕ

margin = (λ d)(col(d) - word1(d))

where col = (λ d)(min0 {n: ℕ | left(move)^n ∈ line})
and   word1 = max0({ n: ℕ | left(move)^n ∈ (word-line) ∧
                                n ≤ col(d)})
and   min0 = min ⊕ { {} ↦ 0 }
and   max0 = max ⊕ { {} ↦ 0 }
```

```
NEWLINE: DOC → DOC

NEWLINE = (λ d) spaces(ins(nl)(d))

          where spaces = ins(sp)^margin(d)
```

Unless otherwise specified we have used standard mathematical notation throughout this document, placing ourselves firmly in the framework of modern Set Theory. A more complete definition of the notation, together with the rules of inference of our logic is given in [Sufrin81a].

The set of total functions between two sets -- X and Y -- is written:

$$X \rightarrow Y$$

and is a subset of the set of partial functions, which is written:

$$X \nrightarrow Y$$

This in turn is a subset of the set of relations, which is written:

$$X \leftrightarrow Y$$

The set of relations is modelled by the set of all subsets of the cross-product:

$$X \times Y$$

in other words:

$$(X \leftrightarrow Y) = \mathbb{P}(X \times Y)$$

The expression $\mathbb{P}(X)$ denotes the powerset of a given set, X, *ie* the set of all its subsets; $\mathbb{F}(X)$ denotes the set of all *finite* subsets of X; $\mathbb{N}$ denotes the set of natural numbers, 0, 1, 2, ... The *domain* of a relation (or function), r, is written $\mathrm{dom}(r)$ and its range is written $\mathrm{ran}(r)$. The identity function on X is written $\mathrm{id}(X)$.

Generic Definitions

In our notation, definitions and specifications may be given *generically*. This is a common practice in mathematics which has not yet been honoured with a conventional notation, but which can usually be recognised in mathematical documents by rhetoric of the form "let X and Y denote arbitrary sets.." and in our notation by the form:

$$X, Y$$

...
...

Useful Combinators

For the sake of brevity and simplicity we give purely axiomatic specifications for *functional* arguments of the combinators we have used: in fact these are just special cases of their definitions for *relations*. Constructive definitions for relational combinators which satisfy the axioms are given elsewhere [Abrial80], [Sufrin81a].

Functions (and relations) are introduced by a *signature* which gives their mathematical type followed by *axioms* which give their properties. For example, composition (of partial functions).

X, Y, Z

$$\circ : (Y \nrightarrow Z) \times (X \nrightarrow Y) \rightarrow (X \nrightarrow Z)$$

$$(\forall\, g: Y \nrightarrow Z;\ f: X \nrightarrow Y)\quad (\forall\, x: X \mid x \in dom(f) \wedge f(x) \in dom(g))\quad (f \circ g)(x) = f(g(x))$$

A reasonable syntactic rule of thumb is that functions and relations with alphanumeric names will be used as prefixes: those with underlined alphanumeric names and those whose names are otherwise formed will be used as infixes. The defining axiom usually sets the tone.

When the exact generic type of a function or relation cannot be determined from the types of its arguments this may be given in the following manner:

```
foo ∘[P,Q,R] baz
```

in which the function `foo` is composed with the function `baz` by the composition operator of functionality:

$$(Q \nrightarrow R \times P \nrightarrow Q) \rightarrow P \nrightarrow R$$

In fact the need for this hardly ever arises, and we usually omit all actual generic parameters (`[P,Q,R]` above) when writing the names of functions.

The domain restriction operator maps a function, f and a subset S of elements to a function which agrees with f on the set S and is elsewhere undefined.

X, Y

$$\upharpoonright : ((X \nrightarrow Y) \times \mathbb{P}(X)) \rightarrow (X \nrightarrow Y)$$

$$(\forall\, f: X \nrightarrow Y;\ S: \mathbb{P}(X))\quad dom(f \upharpoonright S) = dom(f) \cap S\quad (\forall\, x: X \mid x \in dom(f \upharpoonright S))\ (f \upharpoonright S)(x) = f(x)$$

The functional overriding operator maps a pair of functions to one which agrees with the first everywhere except on the domain of the second.

X, Y

$\oplus : (X \leftrightarrow Y) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y)$		
$(\forall f, g : X \leftrightarrow Y;\ x : X)$		
$x \in dom(g)$	$\Rightarrow$	$(f \oplus g)(x) = g(x)$
$x \notin dom(g) \wedge x \in dom(f)$	$\Rightarrow$	$(f \oplus g)(x) = f(x)$
$x \notin dom(g) \wedge x \notin dom(f)$	$\Rightarrow$	$x \notin dom(f \oplus g)$

A function may be mapped to one whose domain lacks a certain set of elements by the domain contraction operator.

X, Y

$\backslash : ((X \rightarrow Y) \times \mathbb{P}(X)) \rightarrow (X \rightarrow Y)$
$\backslash = (\lambda f : X \leftrightarrow Y;\ S : \mathbb{P}(X))\ (f \upharpoonright (X - S))$

The above is an example of definition by λ-abstraction. The full canonical form demands the specification of the types of the bound variables but an acceptable abbreviation is

$$\backslash = (\lambda f, S)(f \upharpoonright (X - S))$$

which, in the context of the function's signature, conveys as much information.

Finite Mappings

Functions with finite domains may be defined by giving all the argument-result pairs; for example the function below is of type: $\mathbb{N} \leftrightarrow \mathbb{N}$

$$\{ 1 \mapsto 0;\ 0 \mapsto 1 \}$$

and maps 1 to 0 and 0 to 1.

Iteration of a Function

A homogeneous function (*ie* from X to X) may be mapped by *iteration* into a function of the same type. The iteration operator is specified below in a style reminiscent of a recursive definition.

X

```
^: ((X ⇸ X) × ℕ) → (X ⇸ X))
(∀ f: X⇸X; n: ℕ)
      f^0 = id(X)
      f^(n+1) = f ∘ (f^n)
```

thus

$$f^2 = f \circ (f^1) = f \circ f \circ (f^0) = f \circ f \circ id = f \circ f$$

We use this operator so often that we have a special syntactic sugar for it -- namely superscription:

$$f^{expression} = f\text{^}expression$$

Although this definition is *analogous* to iteration in a programming language it is not the same thing since the domain of an iterated function may be rather restricted. For example

$$dom(\ pred^5\ ) = \{\ n: \mathbb{N} \mid n \geq 5\ \}$$

because pred is defined on the natural numbers only.

Image of a Set through a Function

The *image* of a set of elements S through a function f is defined by:

$$f(\ S\ ) = \{\ y: Y \mid (\exists\ x: X \mid x \in S \wedge y = f(x))\ \}$$

It is the set of all those elements in Y to which f maps an element of S.

The sequences of elements of X are defined generically as a subset of the partial functions from the natural numbers to X. Their domains are finite segments of the natural numbers starting at 1

X

```
seq: P(N ⇸ X)

seq = { f: N⇸X | dom(f)∈P(N) ∧ dom(f)=1..card(dom(f)) }
```

Sequences occur so frequently in specifications that we have a notation constructing them explicitly: the sequences

{ 1↦a; 2↦b; 3↦c } { 1↦FOO }

can be written ⟨a b c⟩ and ⟨FOO⟩; the empty sequence is written: ⟨⟩. The set of nonempty sequences is:

seq1[X] = seq[X] - {⟨⟩}

The most important functions on sequences are specified below. The length function, (#) and the elementary constructor (cons) are defined constructively.

X

```
#:      seq[X] → N
cons:   X × seq[X] → seq[X]

# = (λ s)(card(dom(s)))

cons = (λ x, s)( (s ⨾ suc) ⊕ { 1↦x })
```

The function card maps a finite set into the number of elements it has. thus the length of a sequence is the number of elements in its domain. The sequence constructor "pushes" an element onto the front of a sequence. An example will clarify matters

c cons ⟨a b⟩ =

({ 1↦a; 2↦b } ⨾ suc) ⊕ { 1↦c } =

({ 2↦a; 3↦b }) ⊕ { 1↦c } =

{ 1↦c; 2↦a; 3↦c } =

⟨c a b⟩

The remaining sequence functions can be defined in terms of the basic constructor by axioms reminiscent of *recursive definitions*.

X

```
*:        seq[X] × seq[X]  → seq[X]
reverse:  seq[X] → seq[X]
head:     seq[X] ↛ seq[X]
tail:     seq[X] ↛ seq[X]
first:    seq[X] ↛ X
last:     seq[X] ↛ X
```

```
(∀ s, s1, s2: seq[X]; x: X)
  ⟨⟩ * s = s
  (x cons s1) * s2 = x cons (s1 * s2)

  reverse(⟨⟩) = ⟨⟩
  reverse(x cons s) = reverse(s) * ⟨x⟩

  dom(first) = dom(tail) = seq1[X]
  first(x cons s) = x
  tail(x cons s) = s

  dom(head) = dom(last) = seq1[X]
  last(s * ⟨x⟩) = x
  head(s * ⟨x⟩) = s
```

References

[Abrial 80]
Jean-Raymond Abrial
The Specification Language Z: Basic Library
Specification Group Working Paper
Programming Research Group
Oxford, 1980

[Jones]
C B. Jones
Software Development. a Rigorous Approach
Prentice-Hall International Series in Computer Science
(Series Editor C.A.R. Hoare)
Prentice-Hall International, 1980.

[Sufrin 80a]
Bernard Sufrin
Specification of a Display Editor
Specification Group Working Paper
Programming Research Group
Oxford, 1980

[Sufrin 81a]
Bernard Sufrin
Reading Formal Specifications
Technical Monograph PRG-24
Programming Research Group
Oxford, 1981

[Sufrin 81b]
Bernard Sufrin
Correctness of a Display Editor Implementation
Specification Group Working Paper
Programming Research Group
Oxford, (forthcoming)

PROGRAMMING RESEARCH GROUP TECHNICAL MONOGRAPHS

JUNE 1981

This is a series of technical monographs on topics in the field of computation. Copies may be obtained from the Programming Research Group, (Technical Monographs), 45 Banbury Road, Oxford, OX2 6PE, England.

PRG-1 *(out of print)*

PRG-2 Dana Scott
Outline of a Mathematical Theory of Computation

PRG-3 Dana Scott
The Lattice of Flow Diagrams

PRG-4 *(cancelled)*

PRG-5 Dana Scott
Data Types as Lattices

PRG-6 Dana Scott and Christopher Strachey
Toward a Mathematical Semantics for Computer Languages

PRG-7 Dana Scott
Continuous Lattices

PRG-8 Joseph Stoy and Christopher Strachey
OS6 – an Experimental Operating System for a Small Computer

PRG-9 Christopher Strachey and Joseph Stoy
The Text of OSPub

PRG-10 Christopher Strachey
The Varieties of Programming Language

PRG-11 Christopher Strachey and Christopher P. Wadsworth
Continuations: A Mathematical Semantics for Handling Full Jumps

PRG-12 Peter Mosses
The Mathematical Semantics of Algol 60

PRG-13 Robert Milne
The Formal Semantics of Computer Languages and their Implementations

PRG-14 Shan S. Kuo, Michael H. Linck and Sohrab Saadat
A Guide to Communicating Sequential Processes

PRG-15 Joseph Stoy
The Congruence of Two Programming Language Definitions

PRG-16 C. A. R. Hoare, S. D. Brookes and A. W. Roscoe
A Theory of Communicating Sequential Processes

PRG-17 Andrew P. Black
Report on the Programming Notation 3R

PRG-18 Elizabeth Fielding
The Specification of Abstract Mappings and their Implementation as B^+-trees

PRG-19 Dana Scott
Lectures on a Mathematical Theory of Computation

PRG-20 Zhou Chao Chen and C. A. R. Hoare
Partial Correctness of Communicating Processes and Protocols

PRG-21 Bernard Sufrin
Formal Specification of a Display Editor

PRG-22 C. A. R. Hoare
A Model for Communicating Sequential Processes

PRG-23 C. A. R. Hoare
A Calculus of Total Correctness for Communicating Processes

PRG-24 Bernard Sufrin
Reading Formal Specifications