FROM Z TO C : ILLUSTRATION OF A RIGOROUS DEVELOPMENT METHOD

by

D.S. Neilson

Technical Monograph PRG-101
ISBN 0-902928-78-3

Hilary 1990

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

From Z To C:

Illustration Of A Rigorous Development Method

D. S. Neilson

Wolfson College, Oxford

From Z To C:

Illustration Of A

Rigorous Development Method

Contents

PART ONE

Overview

Introduction

One of the most important developments in computer science has been that of data abstraction, which has enabled the specification of computer systems without regard to implementation detail, and has resulted in the development of many formal specification languages, for example [17], [28].

A formal specification will undoubtedly provide for a greater understanding of a computer system [22]. It should not, however, be regarded as an end in itself: it may be regarded as a contract between the customer and implementor of a system, and must ultimately be judged on whether it results in the production of better quality software or in the more efficient production of software [15].

Consequently, the process of transforming a formal specification into executable code has attracted attention of many theoreticians, and early work [12], [33] has more recently been supplemented by the production of refinement calculi [2], [5], [7], [14], [21], [23].

In the literature the examples illustrating the refinement techniques are, of necessity, modest in scope and size; usually the specification is presented in a single flat development upon which the method is then demonstrated.

Little investigation has been done into putting such theoretical results into practice in realistically-sized applications (examples are surveyed below). The main purpose of our work is to demonstrate that a suitable formal basis can be practically and usefully employed in the development of "real" software.

The vehicle for our illustration is a full screen text editor. Whilst this may not be regarded as a commercial-scale development, it is of a size (the specification comprises nearly ninety pages) sufficient to enable conclusions to be made regarding the "scaling-up" of the method to much larger applications.

We had to develop a technique that would cope with the problems of size, and the key factor in the development method is its hierarchical nature, enabling the refinement to proceed in manageable parts. The abstract state is composed of approximately thirty components having over twenty five invariant relationships, with the implementation comprising approximately the same. and consideration must be given to each of these constituent for each of the sixty operations that are specified and implemented. Even with a specification of modest size, the problems of complexity are considerable.

We propose a novel hierarchical approach to the specification/refinement process. We start with a simple mathematical model of the system and embellish this model in a series of steps (a hierarchy of levels, each one isolating and treating a particular aspect of the requirements) in such a way that each new level embeds the previous one. This specification style is illustrated in [8]. Note that we use the term "hierarchy" to mean a single level as well as the more normal meaning of a set of levels.

In such a specification only the top-level hierarchy will completely define the desired system. However intermediate levels will also be of significance, since it is possible to apply a refinement calculus at any level to produce a fully deterministic and implementable concrete data type corresponding to each hierarchy (or abstract data type) of the specification. The modularity so-achieved will make reasoning about the development a more manageable task.

It is, of course, possible that this approach might result in an inefficient implementation. since the structure of the design will not necessarily be compatible with that of the specification. but we feel that program transformation techniques [26] can remedy the situation (and, see our conclusions below).

A further advantage of the method concerns prototyping. A crucial problem in constructing any formal specification is to ensure correspondence with an initial (usually informal) set of requirements. One solution that has been proposed is to write the specification in an executable language [11], [31] enabling the specification to be tested as it is written. The nature of such declarative formulations, however, tend to be make them more difficult to read than those written in a non-executable specification language (since the latter need not provide the algorithmic solution that the former, by definition, requires) and necessarily compromises the data abstraction qualities of the specification process.

Whilst we support the view that a formal specification should contain a body of theory to help build confidence that it does indeed describe the informal model that it is meant to, we also feel that a rapid prototyping facility would considerably aid this task.

The refinement technique that we use provides for a complete implementation of a specification hierarchy, and this serves the same purpose as a rapid prototype, since we are able to test the specification against the requirements at each stage of the development. The benefit of our approach is that the code produced is not discarded: it forms an integral part of the final implementation.

The emergence of data abstraction is clearly advantageous in many respects (for example, allowing attention to be focused on the similarities of data types, rather than contrasting their differences) but it is not without its problems. When a specification is implemented it will be on hardware that does not have an infinite supply of resources, and although we do not wish to compromise the abstraction process by the inclusion of such considerations, they are of paramount importance in the implementation.

It is the job of the refinement calculus to bridge the gap between an abstract specification and its implementation: the calculus that we present extends the concept of refinement to permit the introduction of resource limit considerations by consideration of "acceptably" inadequate design decisions.

Of necessity this is a long, detailed and technical piece of work, which we present in four main parts. In this part (Part 1) we state our aims, conclusions and related comments. In Part 2 we present the refinement calculus which provides the formal basis that underpins our development method. We present the abstract specification of the editor in Part 3, and its refinement in Part 4. Appendices A and B summarise the hierarchies of Parts 3 and 4 respectively. The implementation is given in Appendix C.

Each part is, largely, self-contained, with its own introduction and contents sections. The advice we would give to the reader wishing to consider one particular part would be to start with its introduction, followed by the relevant appendix (in the case of Parts 3 or 4, to give an overall picture), before proceeding with the detail.

We assume a knowledge of **Z** [27], [28], [30] and the Schema Calculus [19], [20]. The numbering of definitions etc. is best explained by example: Lemma 3:1.4b refers to the second lemma appearing Section 1.4 of Part 3.

The size of the project has dictated our methodology and also affected our presentation. Although we are able to give the complete abstract specification of the editor, in order to keep the thesis down to a reasonable size, we do not present the refinement in full. We have been honest, however: the entire refinement has been developed rigorously in the manner that we illustrate.

The stimulus for the project was provided by [32] which closely followed the structure of the specification on which it was based [29]; the specification was presented in a hierarchy of three levels, and the implementation was similarly constructed with each level embedded in the next. Although the derivation of the implementation was completely informal, it was felt that the control achieved by using that structure was considerably greater than would otherwise have been the case.

We chose our implementation language, C [18], mainly for its speed and since it was readily available. Other high-level languages would have served the purpose equally well since the programming constructs that our refinement calculus requires (assignment, sequencing, "if" and "do" [5], [7]) are always provided. It is worth noting that languages more strongly typed than C would not provide a "safer" implementation: we place no type-checking requirement on the programming language.

Conclusions And Further Work

The accent on our approach throughout this project is on rigour rather than formality: for example, we indicate which rules are applicable rather than proving that the rules apply. However the development method does permit a completely formal derivation by virtue of the refinement calculus given in Part 2.

Our experience suggests that a lesser degree of formality could, where it was felt to be necessary, be adopted: each abstract data type will usually contain many operations, some of which will be broadly similar, and once one of a set of such operations has been refined to code, that for the other operations may reasonably safely be written down without recourse to the refinement calculus. Of course, such informality will result in code requiring thorough testing.

The abstract specification played a crucial role in ensuring a deep understanding of all aspects of the system. This has been the experience of many others (for example, [22]).

The development of the six chosen refinement hierarchies proceeded remarkably smoothly, the transition of the operations from specification to code presenting few problems. We recognize that even working within a completely formal framework of program development will not automatically ensure bug-free code. The errors that occurred in our implementation, however, have been of a trivial nature (typing errors and the like) and there have been no errors of a "serious" nature (requiring the re-writing of large parts of the implementation). Of course we have re-written parts of the implementation (there are many possible refinements of a given specification) in the quest for improvement, and the modular structure of the implementation has made this task easier than it would otherwise have been.

In order to simplify the description of operations involving i/o, we included a brief specification of our understanding of some operating system and terminal hardware operations (orthogonal to the main model). Although this formal statement of these operations was of considerable use in the construction of the editor interface, both in the specification and refinement phases, we estimate that interfacing the editor took at least fifty percent of the time spent on the implementation!

Whilst some of this time may be explained by our programming inexperience, the inherent problem of formal development within an operating system and hardware environment that is almost exclusively informal is considerable, and much investigation remains to be done.

The facility to test the specification against its requirements proved to be extremely valuable. Although it didn't uncover any major disparity, it did give rise to some fine-tuning of our requirements, the main one concerning cursor movement, and the orthogonal development of the *QP* state (Part 3, Section 4).

We did, in fact, partially implement the display of the editor (the *Doc9* hierarchy, Part 3, Section 8) at an early stage so that we could view lower-level hierarchies. This approach is made possible because of the independence of implementation hierarchies due to the "report-passing" style of programming adopted (see Part 4, Section 0).

We completed the specification before starting its refinement, and in some cases the (minor) changes made as a result of testing did percolate up through the specification hierarchies, which leads us to believe that for some programs, the simultaneous development of specification and implementation might be a very useful approach - the hierarchy under current construction will then be built on lower levels which are demonstrated to have met the informal requirements (through their implementation), minimising specification change.

A further advantage in adopting a simultaneous development strategy is that the development team would, at regular intervals, have something concrete for discussion with the client, permitting early feedback whilst also demonstrating that progress is being made!

The consideration of specification change is important since, although we can demonstrate to the client that the specification matches his/her requirements, the change may be forced by factors outside their immediate control.

We wished to demonstrate the impact of specification change on the refinement method by implementing a specification that we would then change and re-implement. In fact the specification that we present is the amended one; initially we did not permit the use of regular expressions in the search operations, and did not include the move-to-line operation (Part 3, sections 7.2 and 6.4.6 respectively). Space considerations prevent us from presenting the original specification and refinement.

The introduction of regular expressions necessitated the re-refinement of the two search operations (resulting in the introduction of the *CharMatched* routine of *ConcDoc9* (Appendix C, page xi). The addition of the move-to-line command necessitated its refinement in *ConcDoc8* (Appendix C, page xiii), but it could then be promoted in the same way as other operations on that state, and required no further work.

Further, after the implementation was complete we changed our notion of a line in the lowest-level hierarchy (we initially modeled a document line *Doc1* and a display line *Doc3* in the same way, and realised, at a very late stage, that they would be better modelled in a different way). The scope of this specification change was limited to the four line operations in *Doc1*, and, accordingly, our only change in the refinement concerned those four operations.

Clearly the hierarchical structure of the method limits the amount of work that has to be done as a result of specification change, considerably simplifying the task of software maintenance.

Another aspect of the modularity of the development method that we were not able to investigate, but feel that it would be worthwhile to do so, is the re-usability of the specification.

We feel it would be possible to add and refine specification hierarchies, in exactly the same way that we have done here, using the existing code; again, the "report-passing" implementation facilitates this approach.

One example of re-use would be to regard the editor as the basis on which, say, a functional programming tool was to be built: by removing the display *Doc9* hierarchy (Part 3, Section 8), adding hierarchies to provide the necessary functionality (e.g. the addition of "fold" and "unfold" operations), promoting existing operations to the new hierarchies, and putting the display module back at the highest level would enable exactly the same method of implementation that we have used to be followed. The existing code would form an integral part of the extended implementation.

Critics of formal methods will point to the impossibly large number of proof obligations associated with any reasonably-sized program, and this has been the main reason for our rigorous, rather than formal, treatment.

Although we found the power of the Schema Calculus to be a considerable asset in the construction of the specification, but would welcome a tool for automatic schema expansion, the repeated use of schema inclusion in the construction of the specification means, particularly at higher levels, that the problems associated with the identification of proof obligations are severely compounded.

Since the completion of this thesis we have employed a proof-assistant [1], to identify and discharge the obligations associated with the *Doc1* specification hierarchy (Part 3, Section 1). Over one hundred proof obligations emerged. It is clear that any formal development of a large system without the aid of machine assistance would present considerable problems.

We strongly feel that there is a clear need for machine assistance, both in the identification and discharge of proof obligations. Parts of the refinement can be calculated and here, also, computer assistance would be most welcome. The provision of a support environment, for example, as described in [4], would certainly make the entire process more manageable and would, we feel, enable a more formal and less rigorous approach to be adopted.

We had anticipated that the performance of the editor would be inadequate, and, as indicated in the introductory section, that some program transformation would be necessary. To our pleasant surprise, however, we found that the editor's response times certainly matched that of the one that formed the basis for its requirements [32], and consequently we have left the implementation in a structure that exactly matches that of its specification. We have no reason to believe that our code is less efficient than would have been produced by more traditional means.

My design/programming inexperience was a major contribution to the duration of the project: correct refinement does not necessarily imply a good design! It is argued that intuition and experience are a computer scientist's most valuable tools, and any techniques used may, at best, be a supplement to them [24]. We feel, however, that there is considerable room for creativity in methodologies such as the one we present, and indeed that a design team would welcome a basis that enabled the determination that a particular implementation did exactly the job for which it was designed.

Related Work

As stated in our introdnction, there are very few examples of large- and medium-scale formal development of software; this applied science is still very much in its infancy. Much interest has centered on "safety-critical" software (for example where peoples' lives may he endangered by software failnre), bnt the nature of these projects is snch that pnblication is restricted.

The Vienna Development Method, VDM [17], is the longest-established of the formal development methods, and has been used extensively in both academic and industrial courses. It is probahly not now as widely used as other specification langnages (notably **Z**), but it has been of fundamental importance in its influence of formal methods techniques.

VDM has been applied most notably in the areas of systems programming, where the complexity of the code is particularly suited to formalism, and programming language semantics, most notably the description of PL/1.

It is noted that the scale of such developmenrs often renders the work nnsuitable for normal publication [17], and if such work is to gain a wider audience than at present it is an area that clearly needs urgent attention.

Since 1984, IBM have been using formal methods in the development of CICS (Customer Information Control System) [16]. It is a large transaction processing system (comprising over 800,000 lines of code) and existed before the introduction of formal development, the later being nsed in the production new, rather than existing, modules.

The method is based on **Z** and the guarded command langnage; much emphasis is placed on the specification phase, which is used as the vehicle for discussion between the design team and the customer (the business planning/ technical sections of the company). Once the specification is agreed it becomes a record of commitment to be fulfilled by the development team.

The refinement of the specification into code is informal and achieved in two stages: a high-level Z document is first produced stating how the design will be implemented, followed by a low-level document written in the guarded command language. The code is written directly from the latter.

Refinement takes the form of "condensing out the simple parts immediately into guarded command langnage, and specifying the more complicated as schemas to be refined further". Experienced programmers are used and mathematical techniques (e.g. the use of loop invariants) employed only when the specification is complex.

In general, code is prodnced at a point at which it is felt to be "safe", and experienced programmers are found to be indispensable. Module testing is then performed before handing over to other groups for system testiug. The specification is fonnd to be invaluable as a reference document at this point.

The main benefits derived from the approach are felt to be that a greater understanding of the problem is achieved, enabling the team to "get it right" at an early point in the process, increased productivity, improved documentation, and the ability of newcomers to the project to come to terms with the problem quickly. Considerable benefits have been identified in the area of specification change, arising out of greater understanding of the functionality of the application. Further, there is strong evidence to suggest that there are fewer bugs in the resulting code than those present using traditional methods.

Much effort has gone into training, with the establishment of an in-house course. It is generally found that it takes a few months for someone, initially having no formal methods training, to become proficient.

The success if the above project has led to further studies in formal development being pursued at IBM [34]. The specification and target languages are again **Z** and that of guarded commands, but the transition from one to the other is on a more formal basis.

Data and operational refinement are treated separately, and the correctness criteria stem from a retrieve relation, and the identification of obligations to be discharged in a similar way to the VDM method (see above).

Emphasis is placed on the tabulation of particular aspects of the development (e.g. preconditions of partial operations), both to minimise errors and to serve as convenient summaries. The developer is also encouraged to review informal checklists at specific stages (e.g. whether or not sufficient use has been made of pre-existing data types). The aim is to provide a standard development method that has a formal basis.

Small-scale applications have proved successful, with the benefits resulting largely paralleling those stated above and it is noted that there is a need for automated assistance and stressed that the creative role of the programmer is not removed.

A formal methods approach has successfully been adopted to develop a floating point arithmetic routine for the transputer [3]. The routines were abstractly specified in **Z**, and the code formally derived with proofs of correctness given to show that it met its specification.

The significance of this project is not in the scale of the application, but in its complete formality and its relation to the hardware aspect of a computer system; it augurs well for the future.

PART TWO

A
Refinement Calculus
For Z

Contents

0 Introduction

We establish a set of rules and proof obligations which will enable the construction of a formal proof that a proposed implementation of an abstract specification meets (i.e. is correct with respect to) its specification. We follow the approach of [17] in separating refinement of data from that of operations.

In Section 1 we consider data refinement on an abstract state (the implementation of a concrete data type for the abstract data type of the specification). We refer to this process as the taking the "design decision" since we are designing a data structure that can be implemented in a programming language. In fact, the change from abstract to concrete data types need not be accomplished in one step, but each step may be regarded as refining to a "more" concrete type, in the sense that the new type is "nearer" to being able to be implemented in a programming language; our rules permit such a stepwise approach.

In Section 2 we consider the implications of the design decision on the refinement of individual operations of the specification, establishing a method for calculating the specification of the weakest (i.e. most general) concrete operation corresponding to an abstract operation. We also show how a reconfiguration of the concrete state may be achieved once the design decision has been taken without incurring further proof obligations.

In Section 3, we establish a set of operation refinement rules based on the pre- and post-conditions inherent in the specification. We may then either adopt a *transformational* approach to operational refinement, since each result may be regarded as a correctness-preserving transformation of the operation and may be applied (without proof) in the refinement process, or we may, using our intuition and experience, produce what we *feel* is a refinement of the operation and use the rules to prove that it is so.

The nature of an abstract specification is such that operational considerations such as resource limitations are usually ignored, since their inclusion would detract from the clarity and conciseness of the specification. In Section 4, we show how resource constraints may be admitted into the refinement process: existing theories of refinement do not allow such activity, and we motivate the need for the inclusion of such considerations, and extend our definition of refinement, establishing the proof obligations thereby incurred.

We use the symbol "$\sqsubseteq$" to mean "is refined by" or "can be safely replaced with" in both data and operational refinement.

0.1 A Note Regarding Presentation

In order to aid readability, we use the convention that vertically aligned predicates imply their logical conjunction. Thus:

$$\begin{array}{l} \mathit{pred1} \\ \mathit{pred2} \\ \mathit{pred3} \ \vee \ \mathit{pred4} \end{array}$$

will be equivalent to:

$$\mathit{pred1} \ \wedge \ \mathit{pred2} \ \wedge \ (\mathit{pred3} \ \vee \ \mathit{pred4})$$

and:

$$\begin{array}{llll} \mathit{pred1} & & & \\ & \mathit{pred2} & \Rightarrow & \mathit{pred3} \\ & & & \mathit{pred4} \\ & \mathit{pred5} & & \\ \vee & & & \\ & \mathit{pred6} & & \end{array}$$

will be equivalent to:

$$\mathit{pred1} \ \wedge \ (((\mathit{pred2} \ \Rightarrow \ (\mathit{pred3} \ \wedge \ \mathit{pred4})) \ \wedge \ \mathit{pred5}) \ \vee \ \mathit{pred6})$$

1 Data Refinement On A General Abstract State

Data refinement involves the implementation of a concrete data type to represent an abstract data type. In this section we consider the implementation of the concrete state for the abstract state. We assume a *fully abstract* state, by which we mean that each abstract representation is unique in the sense that for any two states there exists a sequence of operations (defined on the abstract state) which enable the two states to be distinguished (in [17] this property is referred to as "freedom from implementation bias").

1.1 The Concrete-Abstract Invariant Relation

We define a general abstract state which comprises the abstract object $\vec{AS}$ (a list of abstract variable signatures) together with an invariant predicate *absinv*:

$$AbsState \mathrel{\widehat{=}} [\ \vec{AS} \mid absinv\ ]$$

We wish to data refine *AbsState*, and we assume a design decision in which the concrete state comprises the concrete object $\vec{CS}$ (a list of concrete varaible signatures) together with an invariant predicate *concinv*:

$$ConcState \mathrel{\widehat{=}} [\ \vec{CS} \mid concinv\ ]$$

We require that each concrete state has an abstract counterpart: by doing so we considerably simplify proof obligations (obviating the need for existential quantification over abstract states), and note that this requirement does not inhibit our transformation from abstract specification into code (the refinement of the specification, Part 4): since we have a fully abstract representation, each concrete state will be associated with a unique abstract state.

In general, we also require that each abstract state has at least one concrete counterpart (i.e. that the design decision is adequate): without this requirement it is possible to admit a design decision which implements only a very small part of the abstract specification (the extreme case being an empty design decision), which, clearly, will be of limited practical use and is unlikely to satisfy the specifier of the system. This requirement is, however, too strict since the use of non-determinism in the specification may explicitly allow states for which no concrete counterpart is envisaged, the specifier allowing the designer the freedom of choice as to exactly which states are provided in the implementation. Since, when using this technique, the specifier may communicate his/her wishes only by informal means, we place proof obligations on the implementor and pursue this consideration in the next section.

1.1.1 The "Rel" Schemas

The invariant relationship between the abstract and concrete states may be conveniently captured in a schema which is the conjunction of the abstract and concrete states together with a predicate, *invrel*, describing the relationship between the two states:

$$Rel \mathrel{\widehat{=}} [\ AbsState \wedge ConcState \mid invrel\ ]$$

We may express the requirement that each concrete state corresponds to a unique abstract state using this schema:

Definition $[\sqsubseteq\ 2 : 1.1.1a]$

$$\forall\ ConcState\ \bullet\ \exists_1\ AbsState\ \bullet\ Rel$$

■

It is useful to consider two further schemas in which the direction of the relationship is recognized (i.e. abstract to concrete, or concrete to abstract): the first, *DownRel*, relates a before-abstract state to an after-concrete one, and the second, *UpRel*, relates the two states in the reverse direction:

$$DownRel \ \widehat{=}\ Rel[\vec{CS}' / \vec{CS}]$$
$$UpRel \ \widehat{=}\ Rel[\vec{AS}' / \vec{AS}]$$

The concrete representation for a particular abstract state will not, in general, be unique: in fact for each abstract state the design decision will define an equivalence class of concrete configurations, which may be determined by calculating *UpRel* (relating an arbitrary concrete state - through *absinv* - with the abstract state that it represents) composed with *DownRel* (relating that abstract state back to another concrete state), and we define:

$$ConcRel \ \widehat{=}\ UpRel\ ;\ DownRel$$

which expands to give:

```
ConcRel ________________________________
| ΔConcState
|---------------
| ∃ AbsState_o •
|     Rel[AS_o / AS]
|     Rel[AS_o, CS' / AS, CS]
|________________________________________
```

$$\begin{array}{l} ConcRel \\ \hline \Delta ConcState \\ \hline \exists\ AbsState_o\ \bullet \\ \quad Rel[\vec{AS}_o / \vec{AS}] \\ \quad Rel[\vec{AS}_o, \vec{CS}' / \vec{AS}, \vec{CS}] \\ \hline \end{array}$$

from which we obtain:

$$\begin{array}{l} ConcRel \\ \hline \Delta ConcState \\ \hline \exists\ \vec{AS}_o\ \bullet \\ \quad absinv[\vec{AS}_o / \vec{AS}] \\ \quad invrel[\vec{AS}_o / \vec{AS}] \\ \quad invrel[\vec{AS}_o, \vec{CS}' / \vec{AS}, \vec{CS}] \\ \hline \end{array}$$

If [$\sqsubseteq$ 2 : 1.1.1a] is satisfied, the unique existence of $\vec{AS}_o$ satisfying *absinv* (the first predicate) is guaranteed, since the second predicate associates $\vec{AS}_o$ with $\vec{CS}$ through *invrel*. The final two predicates relate $\vec{AS}_o$ to both $\vec{CS}$ and $\vec{CS}'$, and their simplification will define the relation between $\vec{CS}$ and $\vec{CS}'$ and, hence, the concrete state equivalence class.

For example, if *AbsState*, *ConcState* and *Rel* are as follows:

$$AbsState \;\widehat{=}\; [\, A : seq\ \mathbb{N} \mid \# A \leq N \,]$$
$$ConcState \;\widehat{=}\; [\, C : 1 \,..\, N \rightarrow \mathbb{N} \,;\, P : 0 \,..\, N \,]$$
$$Rel \;\widehat{=}\; [\, AbsState \wedge ConcState \mid A = C \text{ for } P \,]$$

where:

$$\_ \text{ for } \_ : seq\ X \times \mathbb{N} \rightarrow seq\ X$$
$$S \text{ for } N = 1 \,..\, N \lhd S$$

We have:

ConcRel

$\Delta ConcState$

$$\begin{array}{l} \exists\ A_o : seq\ \mathbb{N} \bullet \\ \quad \# A_o \leq N \\ \quad A_o = C \text{ for } P \\ \quad A_o = C' \text{ for } P' \end{array}$$

and since we may verify that [$\sqsubseteq$ 2 : 1.1.1a] holds, we eliminate A_o to get:

$$ConcRel \;\widehat{=}\; [\, \Delta ConcState \mid C \text{ for } P = C' \text{ for } P' \,]$$

which defines the equivalence class (in which any two members must have equal pointers, their arrays must agree up to those pointers, but can have any natural number values after their pointers).

We may regard *ConcRel* as the weakest specification for a concrete state reorganising operation, and we pursue this in Section 2.2.

We now return to the question of adequacy, and we may calculate the subset of the abstraction that the design decision implements by considering the above composition in the reverse order: we compute *DownRel* (relating an arbitrary abstract state - through *absinv* - with a concrete state) composed with *UpRel* (relating that concrete state back to an abstract state), and we define:

$$AbsRel \;\widehat{=}\; DownRel \;;\; UpRel$$

which expands to give:

AbsRel

$\Delta AbsState$

$\exists\ ConcState_o \bullet$
$\quad Rel[\vec{CS}_o / \vec{CS}]$
$\quad Rel[\vec{AS}', \vec{CS}_o / \vec{AS}, \vec{CS}]$

and we obtain:

AbsRel

$\Delta AbsState$

$\exists\ \vec{CS}_o \bullet$
$\quad concinv[\vec{CS}_o / \vec{CS}]$
$\quad invrel[\vec{CS}_o / \vec{CS}]$
$\quad invrel[\vec{AS}', \vec{CS}_o / \vec{AS}, \vec{CS}]$

This time the second predicate indicates that $\vec{CS}_o$ will exist only for those abstract states which have been implemented by the design decision, but if $\vec{CS}_o$ does exist the final two predicates relate both $\vec{AS}$ and $\vec{AS}'$ to $\vec{CS}_o$ through *invrel*, and if [$\sqsubseteq$ 2 : 1.1.1*a*] also holds (when the first predicate will be assured), $\vec{AS}$ and $\vec{AS}'$ must be the same, and so *AbsRel* is defined on a no-change state:

AbsRel

$\Xi AbsState$

$\exists\ \vec{CS}_o \bullet$
$\quad concinv[\vec{CS}_o / \vec{CS}]$
$\quad invrel[\vec{CS}_o / \vec{CS}]$

Hence we may interpret *AbsRel* as representing the identity operation on the subset of the abstract state for which concrete states exist; when the predicate part of *AbsRel* is **true** we have an adequate design decision.

Using the above example, we have:

$AbsRel$

$\Xi AbsState$
$\exists\ C_o : 1 \,..\, N \rightarrow \mathbb{N}\ ;\ P_o : 0 \,..\, N \bullet A = C_o \text{ for } P_o$

the predicate part of which is true, and so *AbsRel* is equivalent to $\Xi AbsState$ indicating an adequate design.

When each concrete state corresponds to a unique abstract state, and each abstract state has a representation in the design, *AbsState* can be safely replaced with *ConcState*:

Definition $[\sqsubseteq\ 2 : 1.1.1b]$

$Rel \mathrel{\widehat{=}} [\ AbsState \wedge ConcState \mid absinv\ ]$

$\forall\ ConcState \bullet \exists_1\ AbsState \bullet Rel$

$AbsRel \mathrel{\widehat{=}} \Xi AbsState$

$\vdash$

$AbsState \sqsubseteq ConcState$

■

When the first requirement is satisfied but the second is not (*AbsRel* is not equivalent to $\Xi AbsState$), we require that the designer has good reason for the partial implementation before we allow the concrete state to implement the abstract; we formalise the concept of "good reason" in Section 4, giving an alternative formulation of the above lemma.

2 Data Refinement Of A General Abstract Operation

In this section we consider the implementation of the conrrete operation for a general abstract operation.

We first demonstrate (Section 2.1) that, once the design decision has been taken, the weakest specification of the corresponding concrete operation may be calculated; the before- and after-state of the concrete operation will correspond to the before- and after-state of the abstract operation through the *Rel* schema.

However, efficiency considerations (for example) may dictate that the operation is best effected on a particular configuration of the concrete state. We know (Section 1.1.1) that a concrete state corresponding to an abstract state will not, in general, be unique, and it is therefore possible to transform the operation such that it is defined on the required concrete configuration.

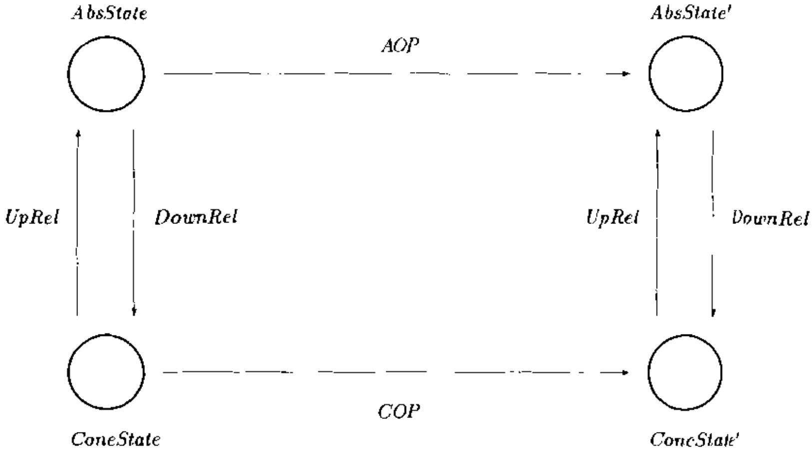
While this may be thought of as a refinement of the concrete operation, the choice of the specific concrete state may be made from the concrete equivalence class defined by the *ConRel* schema, and we choose to regard the process as that of reconfiguration. We show how this may be achieved using *ConcRel* in Section 2.2.

2.1 The Weakest Specification Of The Concrete Operation

The abstract operation *AOP* links a before-state *AbsState* with an after-state *AbsState'*, and we define a general operation whose before- and after-states are related through the predicate *prepost*:

$$AOP \ \widehat{=} \ [\ \Delta AbsState \mid prepost\ ]$$

We assume a design decision with *Rel*, *AbsRel* and *ConcRel* as defined in Section 1. Assuming that the before- and after-states of *AOP* have a representation in the design, we may represent the relationship by the following commuting diagram [10], [12]:



We label the operation linking the concrete states *COP*, and may use the diagram to calculate its weakest (i.e. most general) specification:

$$UpRel\ ;\ AOP\ ;\ DownRel$$

which expands to give:

$$
\begin{array}{|l}
\hline
\textit{UpRel} \,;\, \textit{AOP} \,;\, \textit{DownRel} \\
\hline
\Delta \textit{ConcState} \\
\hline
\exists\ \textit{AbsState}_o, \textit{AbsState}_1 \bullet \\
\quad \textit{Rel}[\vec{AS}_o / \vec{AS}] \\
\quad \textit{AOP}[\vec{AS}_o, \vec{AS}_1 / \vec{AS}, \vec{AS}'] \\
\quad \textit{Rel}[\vec{AS}_1, \vec{CS}' / \vec{AS}, \vec{CS}] \\
\hline
\end{array}
$$

from which we get:

$$
\begin{array}{|l}
\hline
\textit{UpRel} \,;\, \textit{AOP} \,;\, \textit{DownRel} \\
\hline
\Delta \textit{ConcState} \\
\hline
\exists\ \vec{AS}_o, \vec{AS}_1 \bullet \\
\quad \textit{absinv}[\vec{AS}_o / \vec{AS}] \\
\quad \textit{absinv}[\vec{AS}_1 / \vec{AS}] \\
\quad \textit{invrel}[\vec{AS}_o / \vec{AS}] \\
\quad \textit{invrel}[\vec{AS}_1, \vec{CS}' / \vec{AS}, \vec{CS}] \\
\quad \textit{prepost}[\vec{AS}_o, \vec{AS}_1 / \vec{AS}, \vec{AS}'] \\
\hline
\end{array}
$$

As in the simplification of *ConcRel*, [$\sqsubseteq$ 2 : 1.1.1*a*] ensures the unique existence of both $\vec{AS}_o$ and $\vec{AS}_1$ (satisfying the first two predicates). Since $\vec{AS}_o$ and $\vec{AS}_1$ are related to $\vec{CS}$ and $\vec{CS}'$ through *invrel* respectively (third and fourth predicates), the final predicate indicates that the operation may be obtained by the substitution of abstract variables by their *invrel* concrete counterparts, undashed concrete replacing undashed abstract, and dashed concrete replacing dashed abstract.

Continuing with the example introduced in Section 1.1.1, if we have an abstract operation which returns the length of the sequence A:

$$
\begin{array}{|l}
\hline
\textit{AOP} \\
\hline
\Delta \textit{ConcState} \\
x! : \mathbf{N} \\
\hline
A' = A \\
x! = \# A \\
\hline
\end{array}
$$

we have:

$$
\begin{array}{|l}
\hline
UpRel \mathbin{;} AOP \mathbin{;} DownRel \\
\Delta ConcState \\
\hline
\exists\ A_o, A_I : seq\mathbf{N} \bullet \\
\quad \# A_o \leq N \\
\quad \# A_I \leq N \\
\quad A_0 = C \text{ for } P \\
\quad A_I = C' \text{ for } P' \\
\quad A_0 = A_I \\
\quad x! = \# A_o \\
\hline
\end{array}
$$

Since [$\sqsubseteq 2 : 1.1.1a$] holds, we simplify to get:

$$
\begin{array}{|l}
\hline
UpRel \mathbin{;} AOP \mathbin{;} DownRel \\
\Delta ConcState \\
\hline
C \text{ for } P = C' \text{ for } P' \\
x! = \# (C \text{ for } P) \\
\hline
\end{array}
$$

indicating that when *absinv* expresses each component of the abstract state explicitly in terms of the concrete state, we may obtain the weakest concrete operation from the abstract operation by textual replacement of the abstract component for the concrete component, and provided that [$\sqsubseteq 2 : 1.1.1a$] holds, and the before- and after-abstract states have a representation in the design, we incur no further proof obligations

We denote this weakest specification for the concrete equivalent of *AOP* by *AOP*C, and it represents our starting point for the refinement of an operation:

Lemma [$\sqsubseteq 2 : 2.1a$]

$$Rel \mathrel{\widehat=} [\ AbsState \wedge ConcState \mid absinv\ ]$$
$$\forall\ ConcState \bullet \exists_1\ AbsState \bullet Rel$$
$$\exists\ ConcState, ConcState' \bullet AOP \wedge Rel \wedge Rel'$$
$$\vdash$$
$$AOP \sqsubseteq AOP\mathrm{C}$$

■

2.2 Reconfiguring The Concrete State

In this section we show how it is possible to pursue refinement on the weakest concrete specification for *AOP* that conforms to a particular before-state configuration. We achieve this by pre-sequential composition with an operation that produces the desired configuration.

As discussed in Section 1.1.1, the weakest specification for an operation that reconfigures the concrete state is given by *ConcRel*. That operation is, in fact, an identity for *AOP*$\mathbf{C}$, and we could, therefore, take our starting point for operational refinement as:

$$ConcRel \; ; \; AOP\mathbf{C}$$

and, using the results we establish in Section 3, refine *ConcRel* so that its after-state conforms to the specific concrete configuration that we require. However, we could not refine *AOP* so that its before-state was that configuration since we would be violating the *Domain* condition for refinement [$\sqsubseteq$ 2 : 3.2a] that the pre-condition cannot be narrowed. So if we want to pursue refinement on a particular concrete configuration, our starting point must already embody the configuration required.

Our approach is to define an operation like *ConcRel*, but one producing the required configuration as its after-state, and to define another operation like *AOP*$\mathbf{C}$, but one whose before-state also has the desired configuration; we then show that the sequential composition of the two is a refinement of *AOP*$\mathbf{C}$, thus providing us with an alternative starting point for operational refinement.

These two operations can be calculated in the same way as *ConcRel* and *AOP*, and we start by defining the particular concrete configuration by the addition of the predicate *specific* to the concrete state:

$$ConcState_{specific} \;\widehat{=}\; [\; ConcState \mid specific \;]$$

The relationship between the abstract state and this configuration of the concrete state is given by:

$$Rel_{specific} \;\widehat{=}\; [\; Rel \mid specific \;]$$

We follow the same procedure as in Section 1.1.1, and define schemas giving the relationship direction:

$$\begin{array}{lcl} UpRel_{specific} & \widehat{=} & Rel_{specific}[\vec{AS}' / \vec{AS}] \\ DownRel_{specific} & \widehat{=} & Rel_{specific}[\vec{CS}' / \vec{CS}] \end{array}$$

and define $ConcRel_{specific}$ to be the composition of *UpRel* with $DownRel_{specific}$, and so it relates an arbitrary before-concrete state with an after *specific* concrete state:

$$ConcRel_{specific} \;\widehat{=}\; UpRel \; ; \; DownRel_{specific}$$

which expands to give:

$$
\begin{array}{|l}
\hline
\textit{ConcRel}_{specific} \\
\textit{ConcState} \\
\textit{ConcState}'_{specific} \\
\hline
\exists\ \textit{AbsState}_o \bullet \\
\quad \textit{Rel}[\vec{AS}_o / \vec{AS}] \\
\quad \textit{Rel}_{specific}[\vec{AS}_o, \vec{CS}' / \vec{AS}, \vec{CS}] \\
\hline
\end{array}
$$

We now define $\mathbf{AOPC}_{specific}$:

$$AOPC_{specific} \ \widehat{=}\ \ UpRel_{specific} \ ; AOP \ ; DownRel$$

and it expands to:

$$
\begin{array}{|l}
\hline
\textit{UpRel}_{specific} \ ; \textit{AOP} \ ; \textit{DownRel} \\
\textit{ConcState} \\
\textit{ConcState}'_{specific} \\
\hline
\exists\ \textit{AbsState}_o, \textit{AbsState}_1 \bullet \\
\quad \textit{Rel}_{specific}[\vec{AS}_o / \vec{AS}] \\
\quad \textit{AOP}[\vec{AS}_o, \vec{AS}_1 / \vec{AS}, \vec{AS}'] \\
\quad \textit{Rel}[\vec{AS}_1, \vec{CS}' / \vec{AS}, \vec{CS}] \\
\hline
\end{array}
$$

We may interpret this operation as being obtained from *AOP* by the substitution of the before-abstract state by the corresponding concrete state defined by $Rel_{specific}$, and the after-abstract state by the corresponding concrete state defined by *Rel*

We now show that the sequential composition of $ConcRel_{specific}$ and $AOPC_{specific}$ is equivalent to *AOP*, provided $[\sqsubseteq 2 : 1.1.1a]$ is met and the abstract operation is admitted by the design such that its before-state has a representation through $Rel_{specific}$ and its after-state a representation through *Rel*:

Lemma $[\sqsubseteq\ 2 : 2.2a]$

$$Rel \ \widehat{=}\ [\ AbsState \wedge ConcState \mid absinv\ ]$$
$$Rel_{specific} \ \widehat{=}\ [\ Rel \mid specific\ ]$$
$$\forall\ ConcState \bullet \exists_1\ AbsState \bullet Rel$$
$$\exists\ ConcState, ConcState' \bullet AOP \wedge Rel_{specific} \wedge Rel'$$
$$\vdash$$
$$AOPC \ \widehat{=}\ ConcRel_{specific} \ ; AOPC_{specific}$$

■

Proof

Expanding the schema of the right hand side, we obtain:

$$
\begin{array}{|l}
\hline
ConcRel_{specific} \ ; \ AOPC_{specific} \\
\Delta ConcState \\
\hline
\exists \ ConcState_o \ \bullet \\
\quad \exists \ AbsState_o \ \bullet \\
\quad\quad Rel[\vec{AS}_o / \vec{AS}] \\
\quad\quad Rel_{specific}[\vec{AS}_o, \vec{CS}_o / \vec{AS}, \vec{CS}] \\
\quad\quad \exists \ AbsState_1, AbsState_2 \ \bullet \\
\quad\quad\quad Rel_{specific}[\vec{AS}_1, \vec{CS}_o / \vec{AS}, \vec{CS}] \\
\quad\quad\quad AOP[\vec{AS}_1, \vec{AS}_2 / \vec{AS}, \vec{AS}'] \\
\quad\quad\quad Rel[\vec{AS}_2, \vec{CS}' / \vec{AS}, \vec{CS}] \\
\hline
\end{array}
$$

If $ConcState_o$ exists, then by the third antecedent of the lemma, since both $AbsState_o$ and $AbsState_1$ are associated with $ConcState_o$ (the second and third predicates), they must be the same, and we may simplify the predicate part to:

$$
\left[
\begin{array}{l}
\exists \ ConcState_o \ \bullet \\
\quad \exists \ AbsState_1, AbsState_2 \ \bullet \\
\quad\quad Rel[\vec{AS}_1 / \vec{AS}] \\
\quad\quad Rel_{specific}[\vec{AS}_1, \vec{CS}_o / \vec{AS}, \vec{CS}] \\
\quad\quad AOP[\vec{AS}_1, \vec{AS}_2 / \vec{AS}, \vec{AS}'] \\
\quad\quad Rel[\vec{AS}_2, \vec{CS}' / \vec{AS}, \vec{CS}]
\end{array}
\right.
$$

The first predicate relates $AbsState_1$ to $ConcState$ through Rel, and so the lemma's third antecedent ensures the existence of $AbsState_1$, and (fourth predicate) it corresponds to the before-state of AOP. $ConcState_o$ is related to $AbsState_1$ through $Rel_{specific}$ (second predicate) and therefore the last antecedent of the lemma guarantees the existence of $ConcState_o$, and we may further simplify to:

$$
\left[
\begin{array}{l}
\exists \ AbsState_1, AbsState_2 \ \bullet \\
\quad Rel[\vec{AS}_1 / \vec{AS}] \\
\quad AOP[\vec{AS}_1, \vec{AS}_2 / \vec{AS}, \vec{AS}'] \\
\quad Rel[\vec{AS}_2, \vec{CS}' / \vec{AS}, \vec{CS}]
\end{array}
\right.
$$

which is the same as the left hand side.

■

Using this result, we may combine it with [$\sqsubseteq$ 2 : 2.1a] to get an alternative starting point for operational refinement:

Lemma [$\sqsubseteq$ 2 : 2.2b]

$$Rel \mathrel{\widehat=} [\ AbsState \wedge ConcState \mid absinv\ ]$$
$$Rel_{specific} \mathrel{\widehat=} [\ Rel \mid specific\ ]$$
$$\forall\ ConcState \bullet \exists_1\ AbsState \bullet Rel$$
$$\exists\ ConcState.ConcState' \bullet AOP \wedge Rel_{specific} \wedge Rel'$$
$$\vdash$$
$$AOP \sqsubseteq ConcRel_{specific}\ ;\ AOP\mathbf{C}_{specific}$$

■

Therefore if our concrete operation does not need to be conducted on a special concrete state configuration, our starting point is $AOP\mathbf{C}$; if a particular configuration is required, we may use this lemma alternatively to start with the sequential construct $(ConcRel_{specific}\ ;\ AOP\mathbf{C}_{specific})$ in which the former's after-state and the latter's before-state conform to the required configuration; both may be calculated once the design decision has been taken.

3 Operational Refinement

In this section we define exactly what we mean by refining an operation which is specified in Z using the Schema Calculus: we may then (perhaps using our intuition) produce what we *feel* is a refinement of the operation, and *prove* that it is so. We also establish a series of refinement results allowing an alternative approach: each result may be regarded as a refinement-preserving *transformation* of the operation, and may be applied (without proof) in the refinement process.

3.1 The Logical Basis

We use the laws of logic presented in [9] in our proofs of the refinement theory, and also use the following laws, the first of which follows from Constructive Dilemma.2 with d replaced by c, and the second from that same law with d replaced by a, Generalization and Antisymmetry:

$$(a \Rightarrow b) \Rightarrow ((a \wedge c) \Rightarrow (b \wedge c)) \qquad \text{Constructive Dilemma.3}$$

$$(a \Rightarrow b) \Rightarrow ((a \wedge b) \mathrel{\widehat=} a) \qquad \text{Absorption.3}$$

We use two properties of pre-conditions of operations. Since the pre-condition is calculated by existentially quantifying after-variables, and existential quantification distributes through disjunct, pre does likewise:

$\vdash$ pre.1

$$\mathsf{pre}[\bigvee_{i:1..n}(A_i)] \mathrel{\widehat{=}} \bigvee_{i:1..n}(\mathsf{pre}[A_i])$$

■

In general, the same is not true of conjunction; however the existential quantification of a conjunction implies the conjunction of the existential quantifications, and so, by definition of pre:

$\vdash$ pre.2

$$\mathsf{pre}[\bigwedge_{i:1..n}(A_i)] \Rightarrow \bigwedge_{i:1..n}(\mathsf{pre}[A_i])$$

■

3.2 Refinement Of One Operation By Another

An operation B refines an operation A if it satisfies two conditions: the *Domain* condition, which requires that B must be applicable when A is (although B may be applicable for further states as well), and the *Safety* condition, which requires that when A is applicable the results allowed by B are also allowed by A (although B may do more than A). These requirements have been presented in [17] and elsewhere.

Here, we translate the two requirements to a **Z**/Schema environment, presenting them in the form of a definition from which all subsequent results in this section are derived:

Definition $[\sqsubseteq\ 2 : 3.2a]$

$\vdash$

$A \sqsubseteq B$

$\Leftrightarrow$

$\mathsf{pre}[A] \Rightarrow \mathsf{pre}[B]$ *Domain*

$\mathsf{pre}[A] \wedge B \Rightarrow A$ *Safety*

■

Thus *Domain* condition allows us to weaken the pre-condition and the *Safety* condition allows us to strengthen the post-condition.

A corollary to this definition follows from Unit.3: a total operation may be refined only by another total operation:

Corollary $[\sqsubseteq\ 2:3.2b]$

$\textsf{pre}[A] \equiv \textsf{true}$

$\vdash$

$A \sqsubseteq B$

$\Leftrightarrow$

$\textsf{pre}[B] \equiv \textsf{true}$ *Domain*

$B \Rightarrow A$ *Safety*

■

The ordering is transitive, enabling a *stepwise* approach to refinement to be adopted:

Lemma $[\sqsubseteq\ 2:3.2c]$

$(A \sqsubseteq B) \wedge (B \sqsubseteq C)$

$\vdash$

$A \sqsubseteq C$

■

Proof

Domain

Follows from the transitivity of "$\Rightarrow$"

Safety:

1.	$\textsf{pre}[B] \wedge C \Rightarrow B$	$B \sqsubseteq C$
2.	$\textsf{pre}[B] \wedge C \wedge \textsf{pre}[A] \Rightarrow B \wedge \textsf{pre}[A]$	1., Specialization
3.	$\textsf{pre}[A] \wedge B \Rightarrow A$	$A \sqsubseteq B$
4.	$\textsf{pre}[B] \wedge C \wedge \textsf{pre}[A] \Rightarrow A$	2., 3.
5.	$\textsf{pre}[A] \Rightarrow \textsf{pre}[B]$	$A \sqsubseteq B$
6.	$\textsf{pre}[A] \wedge \textsf{pre}[B] \mathrel{\widehat{=}} \textsf{pre}[A]$	5., Absorption.3
7.	$\textsf{pre}[A] \wedge C \Rightarrow A$	4., 6.

■

The ordering is reflexive (recall that $A \mathrel{\widehat{=}} B$ if they are applicable for exactly the same set of states, and the results produced by one imply, and are implied by, the results produced by the other):

Lemma [$\sqsubseteq$ 2 : 3.2d]

$$(A \sqsubseteq B) \land (B \sqsubseteq A)$$
$$\vdash$$
$$A \mathrel{\widehat=} B$$

■

Proof

$(\mathsf{pre}[A] \Rightarrow \mathsf{pre}[B]) \land (\mathsf{pre}[B] \Rightarrow \mathsf{pre}[A]) \Rightarrow$	Antisymmetry
$\mathsf{pre}[A] \mathrel{\widehat=} \mathsf{pre}[B]$	

$(\mathsf{pre}[A] \land B \Rightarrow A) \land (\mathsf{pre}[B] \land A \Rightarrow B) \Rightarrow$	$\mathsf{pre}[A] \mathrel{\widehat=} \mathsf{pre}[B]$
$(\mathsf{pre}[B] \land B \Rightarrow A) \land (\mathsf{pre}[A] \land A \Rightarrow B) \Rightarrow$	propt. Schema
$(B \Rightarrow A) \land (A \Rightarrow B) \Rightarrow$	Antisymmetry
$A \mathrel{\widehat=} B$	

■

Since $\mathsf{pre}[A]$ is obtained from the schema A by the "hiding" operator, $\mathsf{pre}[A]$ conjoined with A is just A.

We note that since every operation trivially refines itself, [$\sqsubseteq$ 2 : 3.2c] and [$\sqsubseteq$ 2 : 3.2d] imply that "$\sqsubseteq$" is a partial order.

We establish a result which will enable refinement by the addition of predicates: if A is the operation we wish to refine, and B an operation such that the pre-condition of the conjoined operation $(A \land B)$ is the same as that of A, then $(A \land B)$ refines A:

Lemma [$\sqsubseteq$ 2 : 3.2e]

$$\mathsf{pre}[A \land B] \mathrel{\widehat=} \mathsf{pre}[A]$$
$$\vdash$$
$$A \sqsubseteq A \land B$$

■

Proof

Domain

Follows directly from the antecedent, using Antisymmetry.

Safety

1. $A \Rightarrow A$	Reflexive.1
2. $\mathsf{pre}[A] \land A \land B \Rightarrow A$	1., Specialization

■

We give a result which will be useful when refining an operation which is promoted from one abstract state to another by the use of logical conjunction. Suppose we wish to promote an operation A by logically conjoining it with a promotion schema P, and suppose B refines A. If we can show that B also refines P, then B also refines the promoted operation $(A \wedge P)$:

Lemma $[\sqsubseteq\ 2 : 3.2f]$

$A \sqsubseteq B$

$P \sqsubseteq B$

$\vdash$

$A \wedge P \sqsubseteq B$

■

Proof

Domain

1.	$\mathsf{pre}[A \wedge B] \Rightarrow \mathsf{pre}[A] \wedge \mathsf{pre}[B]$	pre.2
2.	$\mathsf{pre}[A \wedge B] \Rightarrow \mathsf{pre}[B]$	1., Specialization

Safety

1.	$\mathsf{pre}[A] \wedge B \Rightarrow A$	$A \sqsubseteq B$
2.	$\mathsf{pre}[P] \wedge B \Rightarrow P$	$P \sqsubseteq B$
3.	$\mathsf{pre}[A] \wedge \mathsf{pre}[P] \wedge B \Rightarrow A \wedge P$	1., 2., Constructive Dilemma.1
4.	$\mathsf{pre}[A \wedge P] \Rightarrow \mathsf{pre}[A] \wedge \mathsf{pre}[P]$	pre.2
5.	$\mathsf{pre}[A \wedge P] \wedge B \Rightarrow \mathsf{pre}[A] \wedge \mathsf{pre}[P] \wedge B$	4., Constructive Dilemma.3
6.	$\mathsf{pre}[A \wedge P] \wedge B \Rightarrow A \wedge P$	3., 5.

■

3.3 A Disjunction Of Operations

The following result will enable an operation to be split into a disjunct of smaller operations (typically dictated by a composite pre-condition):

Lemma $[\sqsubseteq\ 2 : 3.3a]$

$\mathsf{pre}[A] \Rightarrow \bigvee_{i:1..n} (\mathsf{pre}[B_i])$ *Domain*

$\forall\ i : 1..n \bullet (\mathsf{pre}[A] \wedge B_i \Rightarrow A)$ *Safety*

$\vdash$

$A \sqsubseteq \bigvee_{i:1..n} (B_i)$

■

Proof

We appeal to definition [$\sqsubseteq$ 2 : 3.2*a*], with $(\bigvee_{i:1..n}(B_i))$ replacing B.

Domain

Follows directly from the *Domain* antecedent and pre.1

Safety

$\forall\ i : 1..n \bullet (\text{pre}[A] \wedge B_i \Rightarrow A)$	$\Rightarrow$	Constructive Dilemma.2
$\bigvee_{i:1..n}(\text{pre}[A] \wedge B_i) \Rightarrow A$	$\Rightarrow$	Distributive.2, Idempotent.2
$\text{pre}[A] \wedge \bigvee_{i:1..n}(B_i) \Rightarrow A$		

■

Having refined an operation into a disjunct of operations, we may wish to preserve the structure so-gained and pursue refinement on each disjunct. If an operation A is composed of disjuncts A_i, and each A_i is refined by B_i such that the pre-condition is not changed, then A is refined by the operation composed of the disjuncts B_i:

Lemma [$\sqsubseteq$ 2 : 3.3*b*]

$\forall\ i : 1..n \bullet \text{pre}[A_i] \mathrel{\widehat=} \text{pre}[B_i]$ *Domain*

$\forall\ i : 1..n \bullet A_i \sqsubseteq B_i$ *Safety*

$\vdash$

$\bigvee_{i:1..n}(A_i) \sqsubseteq \bigvee_{i:1..n}(B_i)$

■

Proof

We again use definition [$\sqsubseteq$ 2 : 3.2*a*], substituting $\bigvee_{i:1..n}(A_i)$ and $\bigvee_{i:1..n}(B_i)$ for A and B respectively.

Domain

$\text{pre}[\bigvee_{i:1..n}(A_i)]$	$\mathrel{\widehat=}$	pre.1
$\bigvee_{i:1..n}(\text{pre}[A_i])$	$\mathrel{\widehat=}$	$\text{pre}[A_i] \mathrel{\widehat=} \text{pre}[B_i]$
$\bigvee_{i:1..n}(\text{pre}[B_i])$	$\mathrel{\widehat=}$	pre.1
$\text{pre}[\bigvee_{i:1..n}(B_i)]$		

Safety

$(\text{pre}[\bigvee_{i:1..n}(A_i)]) \wedge (\bigvee_{i:1..n}(B_i))$	$\mathrel{\widehat=}$	pre.1, Constructive Dilemma.3
$(\bigvee_{i:1..n}(\text{pre}[A_i])) \wedge (\bigvee_{i:1..n}(B_i))$	$\mathrel{\widehat=}$	$\text{pre}[A_i] \mathrel{\widehat=} \text{pre}[B_i]$
$(\bigvee_{i:1..n}(\text{pre}[B_i])) \wedge (\bigvee_{i:1..n}(B_i))$	$\mathrel{\widehat=}$	propt. Schema
$(\bigvee_{i:1..n}(\text{pre}[B_i] \wedge B_i))$	$\mathrel{\widehat=}$	$\text{pre}[A_i] \mathrel{\widehat=} \text{pre}[B_i]$
$(\bigvee_{i:1..n}(\text{pre}[A_i] \wedge B_i))$	$\Rightarrow$	$A_i \sqsubseteq B_i$, Constructive Dilemma.3
$\bigvee_{i:1..n}(A_i)$		

■

3.4 A Sequentially-Composed Operation

We now consider the decomposition of an operation A into two operations B and C, whose sequential composition will produce a refinement for A: each will, in general, be a simpler operation than A, and will themselves be candidates for such refinement, enabling an operation to be refined into the sequential composition of several smaller ones.

If we consider the sequentially composed operation $(A\ ;\ B)$, the pre-condition for the operation must certainly imply A's pre-condition (by definition of "$;$", and if each state output by A is applicable for B, the pre-condition for $(A\ ;\ B)$ is exactly that of A. As a syntactic sugar, we define $A \leadsto B$ to mean that all states produced by A are applicable for B:

Definition $[\sqsubseteq\ 2 : 3.4a]$

$\vdash$

$$A \leadsto B$$
$$\Leftrightarrow$$
$$\mathsf{pre}[A] \ \widehat{=}\ \mathsf{pre}[A\ ;\ B]$$

■

The first result we establish allows an operation A to be refined by $B\ ;\ C$ provided that B is applicable everywhere that A is, and under A's pre-condition, that all output from B is applicable for C and that the results produced by $B\ ;\ C$ must imply those produced by A:

Lemma $[\sqsubseteq\ 2 : 3.4b]$

$\mathsf{pre}[A] \ \Rightarrow\ \mathsf{pre}[B]$	*Domain1*
$\mathsf{pre}[A] \ \Rightarrow\ (B \leadsto C)$	*Domain2*
$\mathsf{pre}[A] \wedge (B\ ;\ C) \ \Rightarrow\ A$	*Safety*

$\vdash$

$$A \sqsubseteq (B\ ;\ C)$$

■

Proof

We appeal to definition $[\sqsubseteq\ 2 : 3.2a]$, substituting $(B\ ;\ C)$ for B.

The *Safety* condition of $[\sqsubseteq\ 2 : 3.2a]$ follows immediately from the *Safety* antecedent.

We now establish *Domain* of $[\sqsubseteq\ 2 : 3.2a]$:

1.	$\mathsf{pre}[A] \ \Rightarrow\ (\mathsf{pre}[B] \ \widehat{=}\ \mathsf{pre}[B\ ;\ C])$	*Domain2*, $[\sqsubseteq\ 2 : 3.4a]$
2.	$\mathsf{pre}[A] \ \Rightarrow\ \mathsf{pre}[B\ ;\ C]$	1., *Domain1*, trans."$\Rightarrow$"

■

Having shown how an operation may be refined into a sequential composition of other operations, we now consider refinement of such a sequential composition: we wish to refine an $(A ; C)$ by refining A to B and C to D such that $(B ; D)$ refines $(A ; C)$. We establish two lemmas, the first of which considers the refinement of A to B, the second one considering the refinement of C to D, and combine them to produce the required result.

In the proofs we use the shorthand that, for example, S'_A denotes the after-state produced by an operation A that corresponds to a before-state of S (where S must, of course, be in the pre-condition of A).

We establish the first lemma:

Lemma $[\sqsubseteq\ 2 : 3.4c]$

$$A \sqsubseteq B$$
$$\vdash$$
$$(A ; C) \sqsubseteq (B ; C)$$

■

Proof

We discharge the proof by appealing to $[\sqsubseteq\ 2 : 3.4b]$, substituting $(A ; C)$ for A.

We first establish *Domain1* of $[\sqsubseteq\ 2 : 3.4b]$:

1.	$\mathsf{pre}[A ; C] \Rightarrow \mathsf{pre}[A]$	propt."; "
2.	$\mathsf{pre}[A] \Rightarrow \mathsf{pre}[B]$	$A \sqsubseteq B$
3.	$\mathsf{pre}[A ; C] \Rightarrow \mathsf{pre}[B]$	1., 2., trans." ⇒ "

We now establish *Domain2* of $[\sqsubseteq\ 2 : 3.4b]$ by contradiction:

4.	$\mathsf{pre}[A ; C] \Rightarrow \neg(\mathsf{pre}[B] \Rightarrow \mathsf{pre}[B ; C])$	Assumption
5.	$\exists\ S : \mathsf{pre}[A ; C] \bullet (S \Rightarrow \mathsf{pre}[B] \wedge \neg(S \Rightarrow \mathsf{pre}[B ; C]))$	4.
6.	$\exists\ S : \mathsf{pre}[A ; C] \bullet \neg(S'_B \Rightarrow \mathsf{pre}[C])$	5., defn. S'_B
7.	$(S \Rightarrow \mathsf{pre}[A]) \Rightarrow (S'_B \Rightarrow S'_A)$	$A \sqsubseteq B$
8.	$(S \Rightarrow \mathsf{pre}[A ; C]) \Rightarrow (S'_B \Rightarrow S'_A)$	7., 1.
9.	$(S \Rightarrow \mathsf{pre}[A ; C]) \Rightarrow (S'_A \Rightarrow \mathsf{pre}[C])$	8., defn. S'_A
10.	$(S \Rightarrow \mathsf{pre}[A ; C]) \Rightarrow (S'_B \Rightarrow \mathsf{pre}[C])$	8., 9., trans." ⇒ "
11.	$\mathsf{pre}[A ; C] \Rightarrow (\mathsf{pre}[B] \Rightarrow \mathsf{pre}[B ; C])$	4., 6., 10.
12.	$\mathsf{pre}[A ; C] \Rightarrow (B \rightsquigarrow C)$	11. defn." $\rightsquigarrow$ "

We finally establish *Safety* of $[\sqsubseteq\ 2 : 3.4b]$:

13.	$\mathsf{pre}[A ; C] \wedge (B ; C) \Rightarrow \mathsf{pre}[A] \wedge (B ; C)$	1., Constructive Dilemma.3
14.	$\mathsf{pre}[A ; C] \wedge (B ; C) \Rightarrow A$	13., *Safety*

■

We now establish the second lemma:

Lemma [$\sqsubseteq$ 2 : 3.4d]

$C \sqsubseteq D$

$\vdash$

$(A ; C) \sqsubseteq (A ; D)$

■

Proof

We appeal to definition [$\sqsubseteq$ 2 : 3.2a] and establish the *Domain* condition by contradiction:

1.	$\neg\,(\mathsf{pre}[A\,;\,C] \Rightarrow \mathsf{pre}[A\,;\,D])$	Assumption
2.	$\exists\; S : State \;\bullet\; (S \Rightarrow \mathsf{pre}[A\,;\,C] \;\wedge\; \neg\,(S \Rightarrow \mathsf{pre}[A\,;\,D]))$	1.
3.	$\mathsf{pre}[A\,;\,C] \Rightarrow \mathsf{pre}[A]$	propt."; "
4.	$\exists\; S : \mathsf{pre}[A] \;\bullet\; (S \Rightarrow \mathsf{pre}[A\,;\,C] \;\wedge\; \neg\,(S \Rightarrow \mathsf{pre}[A\,;\,D]))$	2., 3., Constructive Dilemma.3
5.	$\exists\; S : \mathsf{pre}[A] \;\bullet\; (S'_A \Rightarrow \mathsf{pre}[C] \;\wedge\; \neg\,(S'_A \Rightarrow \mathsf{pre}[D]))$	defn. S'_A
6.	$\neg\,(\mathsf{pre}[C] \Rightarrow \mathsf{pre}[D])$	5.
7.	$\mathsf{pre}[C] \Rightarrow \mathsf{pre}[D]$	$A \sqsubseteq D$
8.	$\mathsf{pre}[A\,;\,C] \Rightarrow \mathsf{pre}[A\,;\,D]$	1., 6., 7.

We also establish *Safety* by contradiction:

9.	$\neg\,(\mathsf{pre}[A\,;\,C] \;\wedge\; (A\,;\,D) \;\Rightarrow\; A\,;\,C)$	Assumption
10.	$\exists\; S : \mathsf{pre}[A\,;\,C] \;\bullet\; \neg\,(S'_{(A\,;\,D)} \Rightarrow S'_{(A\,;\,C)})$	8., 9.
11.	$\exists\; S : \mathsf{pre}[C] \;\bullet\; \neg\,(S'_D \Rightarrow S'_C)$	7., 10., propt."; "
12.	$\neg\,((S \Rightarrow \mathsf{pre}[C]) \;\Rightarrow\; (S'_D \Rightarrow S'_C))$	11.
13.	$\neg\,(\mathsf{pre}[C] \wedge D \;\Rightarrow\; C)$	12.
14.	$\mathsf{pre}[C] \wedge D \;\Rightarrow\; C$	$C \sqsubseteq D$
15.	$\mathsf{pre}[A\,;\,C] \;\wedge\; (A\,;\,D) \;\Rightarrow\; A\,;\,C$	9., 13., 14.

■

Combining the two lemmas, we obtain the desired result:

Lemma [$\sqsubseteq$ 2 : 3.4e]

$A \sqsubseteq B$

$C \sqsubseteq D$

$\vdash$

$(A ; C) \sqsubseteq (B ; D)$

■

Proof

1.	$A\,;\,C \;\sqsubseteq\; B\,;\,C$	$A \sqsubseteq B$, [$\sqsubseteq$ 2 : 3.4c]
2.	$B\,;\,C \;\sqsubseteq\; B\,;\,D$	$C \sqsubseteq D$, [$\sqsubseteq$ 2 : 3.4d]
3.	$A\,;\,C \;\sqsubseteq\; B\,;\,D$	1., 2., [$\sqsubseteq$ 2 : 3.2c]

■

3.5 Refinement To Program Constructs

We now consider refinement of the specification A to the "**if**...**fi**" and "**do**...**od**" constructs presented in [5], [7] and to the assignment statement.

3.5.1 The Guarded Expression

We first consider refinement of A to the construct $(G \longrightarrow B)$, in which the specification B will be executed only if the guard G, a boolean expression, holds.

We may regard $(G \longrightarrow B)$ as the sequential composition of two operations, the first of which, G_{Ξ}, is applicable only when G holds and does not change the state, and, under A's pre-condition, is total with respect to the second operation, B.

Thus the pre-condition for G_{Ξ} is G:

$$\mathsf{pre}[G_{\Xi}] \mathrel{\widehat{=}} G \qquad G_{\Xi}.1$$

When G implies the pre-condition for B under A's pre-condition, G_{Ξ} must do likewise:

$$\mathsf{pre}[A] \Rightarrow ((G \Rightarrow \mathsf{pre}[B]) \Rightarrow (G_{\Xi} \rightsquigarrow B)) \qquad G_{\Xi}.2$$

and since under A's pre-condition G_{Ξ} is total with respect to B, and G_{Ξ} does not change anything, G_{Ξ} sequentially composed with B is just B:

$$\mathsf{pre}[A] \Rightarrow (G_{\Xi} \;;\; B \mathrel{\widehat{=}} B) \qquad G_{\Xi}.3$$

Thus, if A is refined by $(G \longrightarrow B)$, we require that A is total with respect to G, and that under A's pre-condition, B is applicable when G holds and the results produced by B imply those produced by A:

Lemma $[\sqsubseteq\ 2 : 3.5.1a]$

$\mathsf{pre}[A] \Rightarrow G$ *Domain1*
$\mathsf{pre}[A] \wedge G \Rightarrow \mathsf{pre}[B]$ *Domain2*
$\mathsf{pre}[A] \wedge B \Rightarrow A$ *Safety*

$\vdash$

$A \sqsubseteq (G \longrightarrow B)$

■

Proof

To discharge the proof we appeal to $[\sqsubseteq\ 2 : 3.4b]$, substituting $(G_{\Xi} \;;\; B)$ for $(B \;;\; C)$.

We first establish *Domain1* of $[\sqsubseteq\ 2 : 3.4b]$:

1.	$\mathsf{pre}[A] \;\Rightarrow\; G$	*Domain1*
2.	$\mathsf{pre}[A] \;\Rightarrow\; \mathsf{pre}[G_{\Xi}]$	1., G_{Ξ}.1

We now establish *Domain2* of [$\sqsubseteq$ 2 : 3.4b]:

3.	$\mathsf{pre}[A] \;\Rightarrow\; (G \;\Rightarrow\; \mathsf{pre}[B])$	*Domain2*, Importation
4.	$\mathsf{pre}[A] \;\Rightarrow\; (G_{\Xi} \;\leadsto\; B)$	3., G_{Ξ}.2, trans."$\Rightarrow$"

We finally establish *Safety* of [$\sqsubseteq$ 2 : 3.4b]:

5.	$\mathsf{pre}[A] \;\wedge\; (G_{\Xi}\;;\;B) \;\Rightarrow\; A$	*Safety*, G_{Ξ}.3

■

3.5.2 The Alternative Construct

We wish to refine A by the the "**if** ... **fi**" construct, and may informally interpret the statement:

$$\textbf{if}\ (G_1 \longrightarrow B_1)\ [\!]\ (G_2 \longrightarrow B_2)\ [\!]\ \cdots\ [\!]\ (G_n \longrightarrow B_n)\ \textbf{fi}$$

as " if guard G_i is true carry out B_i, and if not then if guard G_j is true carry out B_j, and if not ...". Clearly the construct will be non-deterministic if more than one of the guards holds, and we require that at least one must hold when A is applicable; we further require that each $(G_i \longrightarrow B_i)$ satisfies the *Domain2* and *Safety* conditions of [$\sqsubseteq$ 2 : 3.5.1a]:

Lemma [$\sqsubseteq$ 2 : 3.5.2a]

$\mathsf{pre}[A] \;\Rightarrow\; \bigvee_{i\,:\,1..n}\,(G_i)$	*Domain1*
$\forall\ i : 1..n\ \bullet\ (\mathsf{pre}[A] \wedge G_i \;\Rightarrow\; \mathsf{pre}[B_i])$	*Domain2*
$\forall\ i : 1..n\ \bullet\ (\mathsf{pre}[A] \wedge G_i \wedge B_i \;\Rightarrow\; A)$	*Safety*

$\vdash$

$$A \sqsubseteq \textbf{if}\ [\!]_{\,i\,:\,1..n}(G_i \longrightarrow B_i)\ \textbf{fi}$$

■

Proof

As in the previous section, we regard each $(G_i \longrightarrow B_i)$ as $(G_{i\Xi}\;;\;B_i)$, and so may consider A as being refined to the disjunct of those n operations, and we discharge the proof of this lemma by appealing to [$\sqsubseteq$ 2 : 3.3a].

We first establish *Domain* of [$\sqsubseteq$ 2 : 3.3a]:

1. $\exists\ k : 1 \mathrel{..} n \bullet \mathsf{pre}[A] \Rightarrow G_k$ *Domain1*

2. $\mathsf{pre}[A] \wedge G_k \Rightarrow \mathsf{pre}[B_k]$ *Domain2*

3. $\mathsf{pre}[A] \Rightarrow \mathsf{pre}[B_k]$ 1., 2., Absorption.3

4. $\mathsf{pre}[A] \Rightarrow \mathsf{pre}[G_{k\Xi}\ ;\ B_k]$ 3., G_Ξ.3

5. $\mathsf{pre}[A] \Rightarrow \bigvee_{i : 1 \mathrel{..} n} \mathsf{pre}[G_{i\Xi}\ ;\ B_i]$ 4., Generalization

We now establish *Safety* of [$\sqsubseteq$ 2 : 3.3*a*]:

6. $\forall\ i : 1 \mathrel{..} n \bullet (\mathsf{pre}[A] \wedge G_i \wedge B_i \Rightarrow A)$ *Safety*

7. $\forall\ i : 1 \mathrel{..} n \bullet (\mathsf{pre}[A] \wedge G_i \wedge (G_{i\Xi}\ ;\ B_i) \Rightarrow A)$ 6., G_Ξ.3

8. $G_i \wedge G_{i\Xi} \mathrel{\widehat=} G_{i\Xi}$ G_Ξ.1, propt. Schema

9. $\forall\ i : 1 \mathrel{..} n \bullet (\mathsf{pre}[A] \wedge (G_{i\Xi}\ ;\ B_i) \Rightarrow A)$ 7., 8.

■

We may informally represent [$\sqsubseteq$ 2 : 3.5.2*a*] by the following checklist:

When the pre-condition for A is satisfied:

- at least one of the guards must hold
- each body must be applicable when its guard holds
- the results allowed by each body when its guard holds must allow those results produced by A.

We now consider the same lemma, but when A is composed of a disjunct of n operations A_i (as is frequently the case in an incrementally-constructed specification), and the pre-condition for each A_i forms the guard G_i: if we ensure that B_i is applicable when G_i holds, and then that the results produced by B_i imply those required by each corresponding A_i, we may refine to the "**if**...**fi**" construct:

Lemma [$\sqsubseteq$ 2 : 3.5.2*b*]

$A \mathrel{\widehat=} \bigvee_{i : 1 \mathrel{..} n} (A_i)$

$\forall\ i : 1 \mathrel{..} n \bullet (\mathsf{pre}[A_i] \mathrel{\widehat=} G_i)$ *Domain1*

$\forall\ i : 1 \mathrel{..} n \bullet (G_i \Rightarrow \mathsf{pre}[B_i])$ *Domain2*

$\forall\ i : 1 \mathrel{..} n \bullet (G_i \wedge B_i \Rightarrow A_i)$ *Safety*

$\vdash$

$A \sqsubseteq \mathbf{if}\ [\!]_{i : 1 \mathrel{..} n} (G_i \longrightarrow B_i)\ \mathbf{fi}$

■

Proof

We appeal to [$\sqsubseteq$ 2 : 3.5.2*a*] and first establish *Domain1*:

1. $\bigvee_{i:1..n} (\text{pre}[A_i]) \cong \bigvee_{i:1..n} (G_i)$ *Domain1*, ConstructiveDilemma.2
2. $\text{pre}[\bigvee_{i:1..n} (A_i)] \cong \bigvee_{i:1..n} (G_i)$ 1., pre.1

We now establish *Domain2* of $[\sqsubseteq\ 2 : 3.5.2a]$:

3. $\forall\ i : 1\ ..\ n\ \bullet\ (\text{pre}[A]\ \wedge\ G_i\ \Rightarrow\ \text{pre}[B_i])$ *Domain2*, Absorption.3

We finally establish *Safety* of $[\sqsubseteq\ 2 : 3.5.2a]$:

4. $\forall\ i : 1\ ..\ n\ \bullet\ (\text{pre}[A]\ \wedge\ G_i\ \wedge\ B_i\ \Rightarrow\ A_i)$ *Safety*, Absorption.3
5. $\forall\ i : 1\ ..\ n\ \bullet\ (\text{pre}[A]\ \wedge\ G_i\ \wedge\ B_i\ \Rightarrow\ \bigvee_{j:1..n} (A_j))$ 4., Generalization

■

3.5.3 The Loop Construct

We wish to refine A by the loop construct:

$$\textbf{do}\ (G \longrightarrow B)\ \textbf{od}$$

which we may informally interpret as "if guard G holds carry out B and then start the construct again; if G does not hold then finish".

An important concept in the proof of loop correctness is that of an invariant [5]: a set of predicates which hold before the loop activates, after each iteration of the loop, and after the loop has terminated.

We denote the invariant by I. It will be defined on the same state as that on which A is specified, *State*, and will usually employ the (fixed) initial values of the state variables. Since we wish to accomplish A, and I must hold after termination of the loop, I must form part of A's pre-condition.

Another important consideration concerning loops is that of demonstrating *total* correctness [5] - i.e. showing that the loop does terminate after a finite number of iterations. In order to do this we introduce a *variant* function which associates each state (satisfying A's pre-condition) with a natural number, and we require each iteration of the loop to reduce its value.

The loop represents a total operation: if G holds then B is executed and if G does not hold then the operation is complete. Thus we may consider the loop construct as:

$$\textbf{DO} \cong \textbf{if}\ (G \longrightarrow (B\ ;\ \textbf{DO}))\ \square\ (\neg G \longrightarrow \Xi State)\ \textbf{fi}$$

We have:

$$\text{pre}[\textbf{DO}] \equiv \text{true} \qquad \text{DO.1}$$

$$\text{pre}[(B \,;\, \textbf{DO})] \mathrel{\hat{=}} \text{B} \qquad \text{DO.2}$$

When A is applicable, and the loop guard G holds, we require that the loop body, B, re-establishes I, and when the loop guard does not hold, we require that the state satisfies A's post-condition. Note that in the formulation of the lemma dashed decorations in the antecedents refer to the after-state after a single iteration of the loop:

Lemma [$\sqsubseteq$ 2 : 3.5.3a]

$$\text{pre}[A] \Rightarrow \text{Variant} \in \mathit{State} \longrightarrow \mathbb{N}$$

$$\text{pre}[A] \wedge G \wedge B \Rightarrow \text{Variant}\,(\mathit{State}') < \text{Variant}\,(\mathit{State}) \qquad \mathit{Variant}$$

$$\text{pre}[A] \wedge I \wedge G \Rightarrow \text{pre}[B] \qquad \mathit{Domain}$$

$$\text{pre}[A] \Rightarrow I \qquad \mathit{Safety1}$$

$$\text{pre}[A] \wedge I \wedge G \wedge B \Rightarrow I' \qquad \mathit{Safety2}$$

$$\text{pre}[A] \wedge I' \wedge \neg G' \Rightarrow A \qquad \mathit{Safety3}$$

$\vdash$

$$A \sqsubseteq \textbf{do}\,(G \longrightarrow B)\,\textbf{od}$$

■

Proof

To discharge this proof we use [$\sqsubseteq$ 2 : 3.5.2a] with n=2 and G_1, G_2, B_1 and B_2 equal to G, $\neg G$, (B; **DO**) and $\Xi State$ respectively.

We first establish *Domain1* of [$\sqsubseteq$ 2 : 3.5.2a]:

1. $G \vee \neg G \equiv \text{true}$ — Excluded Middle
2. $\text{pre}[A] \Rightarrow G \vee \neg G$ — 1., Unit.3

We now establish *Domain2* of [$\sqsubseteq$ 2 : 3.5.2a]:

3. $\text{pre}[A] \mathrel{\hat{=}} \text{pre}[A] \wedge I$ — *Safety1*, Absorption.3
4. $\text{pre}[A] \wedge G \Rightarrow \text{pre}[B]$ — 3., *Domain*
5. $\text{pre}[A] \wedge G \Rightarrow \text{pre}[B \,;\, \textbf{DO}]$ — 4., DO.2
6. $\text{pre}[A] \wedge \neg G \Rightarrow \Xi State$ — $\text{pre}[\Xi State] \equiv \textbf{true}$, Unit.3

and *Domain2* with $n = 1$ is established by 5., and 6. establishes *Domain2* with $n = 2$.

We finally establish *Safety* of [$\sqsubseteq$ 2 : 3.5.2a]:

7. $\text{pre}[A] \wedge G \wedge B \Rightarrow I'$ — 3., *Safety2*
8. $\text{pre}[A] \wedge \neg G' \Rightarrow A$ — 3., *Safety3*
9. $\text{pre}[A] \wedge \neg G' \wedge \Xi State \Rightarrow A$ — 8., Specialization

and 9. establishes the *Safety* condition for $n = 2$

The *Safety* condition for $n = 1$ is established recursively by 7., since it guarantees that each iteration starting with G holding will produce a state which satisfies I and so, under A's pre-condition, 7. will guarantee that I is re-established if G still holds, and if G does not hold, 8. will ensure that A is satisfied. termination is guaranteed by *Variant*

■

We may informally interpret [$\sqsubseteq$ 2 : 3.5.3a] by the following checklist:

- identify an invariant that holds when A is applicable
- identify a non-negative variant function

And when the pre-condition for A (and, hence, the invariant) is satisfied:

- the body of the loop must be applicable when its guard holds
- each iteration of the loop must re-establish the invariant
- each iteration of the loop must decrease the variant function
- when the guard no longer holds, A must be satisfied.

In each refinement to a loop construct, we use $\{$ *condition* $\}$ to indicate that, at that particular point in the operation, *condition* holds. Further, we identify the invariant, guard (negation) condition, and bound function explicitly to aid the proof of correctness of the refinement.

3.5.4 The Assignment Statement

The assignment statement has the form $x := v$ and the statement is executed by evaluating v and storing the resulting value in location x. Thus, assuming v evaluates, assignment can be regarded as the substitution of one value for another, and we may achieve substitution by renaming.

For example, to establish the truth of the predicate:

$$(x > 1 \ \land \ x := x + 1) \ \Rightarrow \ x > 2$$

we would establish the result of the equivalent predicate:

$$x > 1 \ \Rightarrow \ (x > 2)[x + 1 / x]$$

We use exactly the same technique in a Schema environment; if an operation A is defined on a state $S \mathrel{\widehat=} [\ x : \mathbf{N}\ ]$, then:

$$\mathsf{pre}[A] \ \land \ x := v \ \Rightarrow \ A$$

is equivalent to:

$$\mathsf{pre}[A] \ \Rightarrow \ A[v / x']$$

provided we can evaluate v which, in this context, means that v must evaluate to a natural number.

When the state consists of variables $x_1, x_2, \ldots, x_n$:

$$\textsf{pre}[A] \ \wedge \ x_1 := v_1 \ ; \ x_2 := v_2 \ ; \ \ldots \ ; \ x_n := v_n \quad \Rightarrow \quad A$$

is equivalent to:

$$\textsf{pre}[A] \ \Rightarrow \ A[v_1 / x_1'][v_2 / x_2']\ldots[v_n / x_n']$$

provided that each v_i evaluates to a valid component of the state on which A is defined.

Assuming that A is defined on *State*, we have:

Lemma [$\sqsubseteq$ 2 : 3.5.4*a*]

$\textsf{pre}[A] \ \Rightarrow \ \exists \ y_1, y_2, \ldots, y_n : State \ \bullet \ y_1 = v_1 \ \wedge \ y_2 = v_2 \ \wedge \ \ldots \ \wedge \ y_n = v_n$
$\textsf{pre}[A] \ \wedge \ A[v_1 / x_1'][v_2 / x_2']\ldots[v_n / x_n']$

$\vdash$

$A \ \sqsubseteq \ x_1 := v_1 \ ; \ x_2 := v_2 \ ; \ \ldots \ ; \ x_n := v_n$

■

Proof
We use definition [$\sqsubseteq$ 2 : 3.2*a*] and establish its *Domain* condition by the first antecedent since the pre-condition for the assignment statement is the existence of the values v_i. The *Safety* condition is established since it is equivalent to the second antecedent.

■

4 Admitting Resource Constraints Into The Refinement

One of the most important developments in computer science has been that of data abstraction, which enables the specification of computer systems without the need to consider details of the implementation.

An example of such detail is a resource which is limited in some way, and over which the system (and certainly the specifier of a system) has no control. In an abstract model we do not wish to include such factors since such considerations will detract from the clarity and conciseness of the specification: for example, when we specify an operation which consumes a particular resource, we wish our attention to be focused on what we require the system to do before and after that resource has been exhausted, rather than on its explicit identification.

To illustrate the consumption a limited resource, we consider two specifications of an abstract state comprising a sequence of numbers, and an operation which concatenates a number on to the front of the sequence.

In the first we identify the maximum value of the limited resource, *Max*:

$[Max]$

$$AbsState\_1 \;\widehat{=}\; [\; S : seq\ \mathbb{N} \mid \#S \leq Max \;]$$

$$\Delta AbsState\_1 \;\widehat{=}\; AbsState\_1 \wedge AbsState\_1'$$

Add_1

$\Delta AbsState\_1$
$s? : \mathbb{N}$
$rep! : Report$

$\#S < Max$
$S' = \langle s? \rangle \frown S$
$rep! =$ "OK"

$\vee$

$\#S = Max$
$S' = S$
$rep! =$ "Full"

The operation specifies that, if the maximum size of the sequence has not been reached, s is concatenated on to the front of S, but once the maximum size of the sequence has been reached, S will not change.

Suppose we choose to implement the model using a fixed size array, together with a pointer. We may represent this design decision as:

ConcState_1

$Arr : 1\,..\,Max \rightarrow \mathbb{N}$
$ptr : 0\,..\,Max$

We show the concrete-invariant relationship through the schema *Rel*, in which the contents of *Arr* up to *ptr* are equated to S (Section 1.1):

Rel_1

$AbsState\_1$
$ConcState\_1$

$S = Arr$ for ptr

Note that, by definition of for, a *ptr* value of *0* corresponds to S being the empty sequence; further, since *ptr* may not exceed *Max*, for each concrete state satisfying the *ConcState_1* invariant, the schema identifies a unique abstract state satisfying the *AbsState_1* invariant. Thus the relationship from concrete to abstract is functional and total.

We calculate *AbsRel* (Section 1.1.1), which simplifies to give:

$$AbsRel\_1 \quad \widehat{=} \quad \Xi AbsState\_1$$

and so $AbsState \sqsubseteq AbsState$ [$\sqsubseteq$ 2 : 1.1.1b].

We now repeat the specification, but this time without specific reference to *Max*:

$$AbsState\_2 \quad \widehat{=} \quad [\ S : seq\ \mathbf{N}\ ]$$

$$\Delta AbsState\_2 \quad \widehat{=} \quad AbsState\_2 \wedge AbsState\_2'$$

Add_2

$\Delta AbsState\_2$
$s? : \mathbf{N}$
$rep! : Report$

$$\begin{array}{l} \quad S' = \langle s? \rangle \frown S \\ \quad rep! = \text{"OK"} \\ \vee \\ \quad S' = S \\ \quad rep! = \text{"Full"} \end{array}$$

This time the operation non-deterministically specifies that either s should be concatenated on to the front of S, or S should remain unchanged.

We use the same design decision:

$$ConcState\_2 \quad \widehat{=} \quad ConcState\_1$$

and, as before, the concrete-abstract relation is functional and total.

We again calculate *AbsRel*, and, after simplification, get:

$$AbsRel\_2 \quad \widehat{=} \quad [\ \Xi AbsState \mid \# S \leq Max\ ]$$

which indicates an inadequate design decision, since we are only able to implement that subset of the abstraction for which the length of the sequence does not exceed *Max*, and we may not appeal to [$\sqsubseteq$ 2 : 1.1.1b].

Of course the specifier would not expect an adequate implementation in this case: it would have to provide for a sequence of infinite length, since, clearly, the specification allows for arbitrarily large sequences to be constructed. However a design decision which implements S as an empty array and which never allows S to change (i.e. the second disjunct of the operation is always chosen) would be unlikely to meet with the specifier's approval!

Here the specifier is simply avoiding implementation detail by use of non-determinism in the specification, and is communicating his/her wishes informally through the report; we may interpret the specification as "whilst there is available capacity, concatenate s on to the front of S, and when the available capacity is exhausted, do not change S". This style of writing Z specifications is commonplace (for example [8]).

This first approach has two obvious disadvantages: it places an obligation on the specifier to consider details of the implementation (even though he/she may have no knowledge of the proposed implementation) and it results in a more "cluttered" specification. We therefore extend our concept of refinement from [$\sqsubseteq$ 2 : 1.1.1b] to cope with specifications written in the second style.

4.1 ∞-Refinement

We therefore propose a process of refinement in which we take an abstract specification to a partial implementation, containing a set of resource constants, and demonstrate that if such constants could be made large enough, we would be able to effect a full implementation of the specification. In other words, the only reason that we have not been able to fully implement the specification is that we do not have computers big enough, and if we increase the size of the target computer we could accordingly increase the subset of the specification that is implemented.

For example, in the second specification above we would need to show that a suitable natural number N can be found such that for every possible value of the abstract sequence S, the predicate $\# S \leq Max$ is satisfied for each value of Max greater than or equal to N:

$$\forall\ S : seq\ \mathbf{N} \bullet \exists\ N : \mathbf{N} \bullet Max \geq N \Rightarrow \# S \leq Max$$

Generalising this result to a predicate *pred*, we need to show that for each resource limit c_i present in the design, we can find a natural number N_i such that *pred* is satisfied when every value of c_i is greater than or equal to the corresponding N_i:

$$\begin{array}{l} \forall\ c_1, c_2, ..., c_n : ResourceLimit \bullet \\ \quad \exists\ N_1, N_2, ..., N_n : \mathbf{N} \bullet \\ \quad\quad c_1 \geq N_1 \wedge c_2 \geq N_2 \wedge ... \wedge c_n \geq N_n \Rightarrow pred \end{array}$$

for which we use the syntactic sugar:

$$\exists\ c_1, c_2, ..., c_n : ResoureeLimit \bullet \lim_{c_1, c_2, ..., c_n \rightarrow \infty, \infty, ..., \infty} (pred)$$

We accordingly extend our definition of refinement and use the symbol "$\sqsubseteq_\infty$", which is read "refines in the limit" (or "∞-refines"). We assume the set of resource limits, *ResourceLimit* (which would include, for example, *maxint* of Pascal), and have:

[*ResourceLimit*]

Definition [$\sqsubseteq$ 2 : 4.1*a*]

$\vdash$

$A \sqsubseteq_\infty B$

$\Leftrightarrow$

$\exists\ c_1, c_2, ..., c_n : ResourceLimit \bullet \lim_{c_1, c_2, ..., c_n \to \infty, \infty, ..., \infty} (A \sqsubseteq B)$

■

where c_i do not appear in A and will be free in B (since they will be "external" constants). Hence we are concerned with the limiting behaviour of the predicates of B containing such constants.

We now extend the result of [$\sqsubseteq$ 2 : 1.1.1*b*] to deal with (acceptably) inadequate design decisions. We require that, in such decisions, each concrete state corresponds to a unique abstract state, and that resource limit constants c_i can be identified such that, as we allow them to increase indefinitely, in the limit the predicate part of *AbsRel* will be true:

Lemma [$\sqsubseteq$ 2 : 4.1*b*]

$Rel \mathrel{\widehat=} [\ AbsState \wedge ConcState \mid absinv\ ]$

$\forall\ ConcState \bullet \exists_1\ AbsState \bullet Rel$

$c_1, c_2, ..., c_n : ResourceLimit$

$\lim_{c_1, c_2, ..., c_n \to \infty, \infty, ..., \infty} (AbsRel \equiv \mathbf{true})$

$\vdash$

$AbsState \sqsubseteq_\infty ConcState$

■

Proof

We discharge the proof by appealing to [$\sqsubseteq$ 2 : 1.1.1*b*]: the first two antecedents are provided directly by those of this lemma.

This lemma's other two antecedents together with the definition of $\Xi AbsState$ give:

$$\exists\ c_1, c_2, ..., c_n : ResourceLimit \bullet \lim_{c_1, c_2, ..., c_n \to \infty, \infty, ..., \infty} (AbsRel \mathrel{\widehat=} AbsState)$$

and so by [$\sqsubseteq$ 2 : 1.1.1*b*] and the distributivity of "$\exists$", we have:

$$\exists\ c_1, c_2, ..., c_n : ResourceLimit \bullet \lim_{c_1, c_2, ..., c_n \to \infty, \infty, ..., \infty} (AbsState \sqsubseteq ConcState)$$

which, by definition [$\sqsubseteq$ 2 : 4.1*a*] yields the desired result.

■

We place an obligation on the designer to ensure that the requirements of [$\sqsubseteq$ 2 : 1.1.1b] or those of [$\sqsubseteq$ 2 : 4.1b] are met when data refining an abstract specification. Clearly, this obligation will not guarantee that a particular design for a specification will satisfy its specifier, since the latter requirement depends upon the identification of appropriate resource limits, and the contents of the set of *ResourceLimit* are undefined. However, we feel that the obligation does force the designer to consider the adequacy of the proposed implementation by giving due regard to informal communication of the specifier, and this might not be the case were the obligation not present.

PART THREE

Z Specification Of A Full Screen Text Editor

Contents

0 Introduction

We present a formal specification of a screen-oriented text editor, written in **Z** [19], [27], [28] and using the Schema Calculus [20].

The specification is developed in a hierarchical manner such that the abstract data representation being currently specified is an embellishment of the previous one. We start with a simple abstract representation of the editor, successively enriching it until we finally have a complete mathematical model. Each abstract data type (an abstract data representation together with an initialisation and a family of operations) is thus embedded in succeeding hierarchies. This technique ensures that each step isolates a specific problem, enabling clear specification objectives, and supporting the concept of "separation of concerns" [1]. Thus at each hierarchy of the specification we are able to specify additional operations (requiring the enriched state) and "promote" existing operations (i.e. re-specify each one on the embellished state, such that its original characteristics are retained).

Where possible we employ an orthogonal development method: the orthogonal model is constructed outside the main model, the two having no common state. The orthogonal model is subsequently introduced into the main model through logical (schema) conjunction, with the possible addition of an invariant indicating the way in which the two are related. This method leaves the main model uncluttered by, for example, the necessary or desirable establishment of theory.

Further, we use generic constructions, in which families of concepts may be captured in a single definition, enabling a theory on the formal generic parameter to be constructed, rather than having to develop like theories for each actual generic parameter.

Each operation specified is total. Typically an operation is specified in several stages using schema inclusion, conjunction and/or disjunction, and an operation with an inherent pre-condition is made total by disjunction with an "error" schema, in a similar way to that presented in [8].

This specification uses many of the ideas put forward in [25], which in turn was heavily motivated by [29]; the latter was written in **Z** but did not employ the Schema Calculus (which hadn't then been fully developed) and employed "higher order" functions (functions that take other functions as arguments) to achieve a hierarchical structure. We gratefully acknowledge that paper as a source of inspiration for this specification.

We summarize our requirements as the editor [32], together with the facility to move the cursor around the quarter plane in which the document resides (rather than just around the document itself), and the provisos that no line may end in whitespace (other than the current line when the cursor is at the right hand end - otherwise it would be impossible to insert a space character at the end of a line) and that the document may not have trailing empty lines; neither can be detected on the terminal screen. An editor should inspire the confidence of the user [6], and without these latter two requirements a move to the bottom of the document or end of line, for example, could have (literally!)

unforeseen consequences. We extend the quote operations (operations not effected by single keystrokes, but requiring the input of text into a buffer) to include the writing and appending of marked text to a named file, and exclude the exchange marked text with paste buffer operation of the editor [32] (since the latter operation may be accomplished by the former).

0.1 The Hierarchy Of Abstract States

Our first simple abstract representation of the editor is the *Doc1* model (Section 1) in which we use a pair of sequences to capture both the content and the current position of a document, the latter being the point at which changes to the document may be made. This model incorporates equivalent character, word and line "views"; we discuss their relationship and specify a change of current position and the insertion and deletion of text by using the view which is most appropriate.

We model the document's display by first considering an unbounded display specification, *UD*, orthogonal to that of the document specified in *Doc1*, which models the display as a sequence of display lines, incorporating a cursor and a cursor line. We conjoin the two models into *Doc2* (Section 2), in which we give the relationship between the contents and current position of a document to its unbounded display and cursor position.

We then "tidy" the display of the document to the *Doc3* model (Section 3), by requiring that the trailing whitespace/null lines requirements are met.

We define the *QP* state (Section 4) in which cursor movement around the quarter plane is developed orthogonally to the main model, subsequently conjoining *QP* with *Doc3* to give the *Doc4* model, in which we require that the cursors of *UD* and *QP* are equivalent.

Further *Doc* models introduce the remaining edit operations. In Section 5 we specify the text manipulation operations concerned with marking text (*Doc5*), and lifting, cutting and pasting text (*Doc6*). Section 6 is concerned with commands which cannot be activated by a single keystroke - for example commands requiring textual input - the Quote commands (*Doc7*). In Section 7 we specify commands which permit the searching for and replacing of text (*Doc8*).

After all edit operations have been specified, our last development is to introduce a movable window on to the unbounded display of the document, Section 8, and we define the top-level hierarchical state, *Doc9*, which represents our complete mathematical model of the editor.

For convenience, we give a summary of the specification state hierarchies in Appendix A.

0.2 A Note Regarding Specification Convention

We continue to use the convention that vertically aligned predicates imply their logical conjunction (see **Part** 2, Section 0.1).

Also we use the convention that, for example, "*string*" represents the sequence of characters $< s, t, r, i, n, g >$.

The reader should refer to [28] for a full description of the **Z** notation: anything not provided by that library will be defined as it is required.

0.3 Specification Proof Obligations

We consider proof obligations that ensure that a possible implementation for the specification exists (thereby ensuring that we have not specified an unimplementable system).

Each specification representation will include an invariant on the state variables (in simple cases this will be the type of the variables), and our first proof obligation is to show that the initialization operation establishes that invariant. Defining our abstract variables as $\vec{AS}$, our state invariant as abs_inv $\vec{AS}$ and denoting our initialisation as *Init* which has after-state variables $\vec{AS}'$, we may formally define our first proof obligation as:

$$\vdash \qquad \text{PO 0}$$
$$\exists\ \vec{AS}' \bullet Init \land \mathsf{abs\_inv}\ \vec{AS}'$$

For example, if our abstract state comprises two variables a and b, with state invariant that both are natural numbers together with the requirement that a must not exceed b, and the initialisation sets both to zero, our proof obligation is:

$$\vdash$$
$$\exists\ a', b' \bullet a' = 0 \land b' = 0 \land a' \in \mathbb{N} \land b' \in \mathbb{N} \land a' \leq b'$$

simplifying to:

$$0 \in \mathbb{N} \land 0 \leq 0$$

Once the state invariant has been established by the initialization operation, we are obliged to show that each subsequent operation preserves the invariant, assuming the pre-condition of the operation: if operation Op has before-state variables $\vec{AS}$ and after-state variables $\vec{AS}'$, we must show that:

$$\mathsf{abs\_inv}(\vec{AS}) \land \mathsf{pre}[Op]$$
$$\vdash \qquad \text{PO 1}$$
$$\exists\ \vec{AS}' \bullet Op \land \mathsf{abs\_inv}(\vec{AS}')$$

Using the example given above, suppose an operation Op has the pre-condition that a is non-zero and decreases a by 1 leaving b unchanged, our proof obligation is:

$$a \in \mathbf{N} \land b \in \mathbf{N} \land a \leq b \land a \neq 0$$
$$\vdash$$
$$\exists\ a', b' \bullet a' = a - 1 \land b' = b \land a' \in \mathbf{N} \land b' \in \mathbf{N} \land a' \leq b'$$

simplifying to

$$a \in \mathbf{N}_1 \Rightarrow a - 1 \in \mathbf{N}$$
$$a \in \mathbf{N}_1 \land b \in \mathbf{N} \land a \leq b \Rightarrow a - 1 \leq b$$

Notice that since since both a' and b' were explicitly set by the operation, the existential quantifier disappears (by the "one-point" rule). When the operation is non-deterministic, however, this is not the case.

For example by defining the operation on the "Δ" abstract state (for an example, see Section 1.3) the before- and after-invariant is frozen in to the specification of the operation, and any variables not explicitly set are thus allowed to change to any value consistent with the state invariant.

Continuing with the same example, suppose another operation is defined on $\Delta AbsState$ (where $AbsState$ incorporates the state invariant that a and b are natural numbers with a not exceeding b) which has the same pre-condition, decreases a by 1, but leaves b non-deterministic. Our proof obligation is:

$$a \in \mathbf{N} \land b \in \mathbf{N} \land a \leq b \land a \neq 0$$
$$\vdash$$
$$\exists\ a', b' \bullet a' = a - 1 \land a' \in \mathbf{N} \land b' \in \mathbf{N} \land a' \leq b'$$

which simplifies to:

$$a \in \mathbf{N}_1 \Rightarrow \begin{array}{l} a - 1 \in \mathbf{N} \\ \exists\ b' \bullet b' \in \mathbf{N} \land b' \geq a - 1 \end{array}$$

We employ an orthogonal method of specification development, combining the orthogonal models using schema conjunction with (usually) an invariant cementing the relationship between the two states (for an example, see Section 2.2). In order to identify our proof obligation we could expand the conjoined state and the proof obligations outlined above would apply. However, if we have already discharged the obligations for each orthogonal state, we may lighten our proof obligation load.

Suppose a state *AbsStateA* incorporates variables $\vec{AS}$ with invariant A_abs_inv, and another state *AbsStateB* has variables $\vec{BS}$ with invariant B_abs_inv, and we schema-conjoin the states to form *AbsStateC*, adding the invariant C_abs_inv. Thus the conjoined state comprises both sets of variables on which both invariants, as well as the additional invariant, holds.

If an operation Op, defined on $\vec{AS}$ and preserving A abs_inv, is promoted to *AbsStateC* through logical conjunction with $\Delta AbsStateC$, we must show that Op does not violate the invariant of the orthogonally specified state B_abs_inv, by showing the existence of an after state of *AbsStateB* under C abs inv, assuming the pre-condition of the operation:

$$\begin{array}{l} \mathsf{A\_abs\_inv}\ (\vec{AS})\ \wedge\ \mathsf{A\_abs\_inv}\ (\vec{AS}')\ \wedge\ \mathsf{pre}[Op] \\ \vdash \\ \exists\ \vec{BS}'\ \bullet\ Op\ \wedge\ \mathsf{C\_abs\_inv}\ (\vec{AS}', \vec{BS}')\ \wedge\ \mathsf{B\_abs\_inv}\ (\vec{BS}') \end{array} \qquad \text{PO 2}$$

For example, suppose we have:

$$AbsStateA \ \widehat{=}\ [\ a, b : \mathbf{N}\ |\ a \leq b\ ]$$
$$AbsStateB \ \widehat{=}\ [\ s, t : \mathbf{N}\ |\ s \leq t\ ]$$

with operations:

$$OpA \ \widehat{=}\ [\ \Delta AbsStateA\ |\ a' = 1\ \wedge\ b' = 2\ ]$$
$$OpB \ \widehat{=}\ [\ \Delta AbsStateB\ |\ s \neq 0\ \wedge\ s' = s - 1\ \wedge\ t' = s\ ]$$

Suppose we have discharged all proof obligations, and we wish to specify a composite state *AbsStateC*, where:

$$AbsStateC \ \widehat{=}\ [\ AbsStateA\ \wedge\ AbsStateB\ |\ b = s\ ]$$

For *OpA* (which is total, and so its pre-condition is therefore *true*), we must show the existence of an after-state of *AbsStateB*, under the invariant of *AbsStateC*:

$$\begin{array}{l} a \in \mathbf{N}\ \wedge\ b \in \mathbf{N}\ \wedge\ a \leq b\ \wedge\ a' \in \mathbf{N}\ \wedge\ b' \in \mathbf{N}\ \wedge\ a' \leq b' \\ \vdash \\ \exists\ s', t'\ \bullet\ a' = 1\ \wedge\ b' = 2\ \wedge\ b' = s'\ \wedge\ s' \in \mathbf{N}\ \wedge\ t' \in \mathbf{N}\ \wedge\ s' \leq t' \end{array}$$

which simplifies to:

$$\exists\ t'\ \bullet\ t' \in \mathbf{N}\ \wedge\ t' \geq 2$$

For *OpB*, we must show the existence of an after-state of *AbsStateA*, under the invariant of *AbsStateC*:

$$s \in \mathbf{N} \land t \in \mathbf{N} \land s \leq t \land s' \in \mathbf{N} \land t' \in \mathbf{N} \land s' \leq t' \land s \neq 0$$

$$\vdash$$

$$\exists\ a', b' \bullet\ s' = s - 1 \land t' = s \land b' = s' \land a' \in \mathbf{N} \land b' \in \mathbf{N} \land a' \leq b'$$

which simplifies to:

$$s \in \mathbf{N}_1 \Rightarrow s - 1 \in \mathbf{N} \quad \exists\ a' \bullet a' \in \mathbf{N} \land a' \leq s - 1$$

Note that we may achieve the same specification by renaming s to b in *AbsStateB*, in which case *AbsStateC* would require no additional invariant (and so C abs_inv would just be *true*). In this case the same proof obligation applies.

Often the discharge of these proof obligations will be trivial (particularly type obligations); when this is not the case, we give a proof that the obligation is met, or indicate how it may be discharged.

Of course we may choose to do more than just meet the specification proof obligations, and state (or prove) further properties relating to the specification which, we feel, may give further insight into the system, increase our confidence in the model we are building, or demonstrate conformity to our initial (informal) requirements.

1 The Doc1 Model

1.1 The Generic Document

The two essential characteristics of the state of a document being edited are its contents and current position; changes made to the contents of a document will take place at that position. These two characteristics may be captured by representing a document as a pair of sequences, one corresponding to the part of the document preceding, and the other to that part which follows the current position. The contents will thus be the concatenation of the two sequences.

A document may thus be modelled as a pair of sequences of *characters*. However it is often convenient to represent the document as a pair of sequences of *words* or *lines* (for instance when moving the cursor by a word or line at a time, rather than a character at a time). Therefore our initial definitions will use the formal parameter X, which we will instantiate by the sets of actual generic parameters of sequences of characters, words and lines to give three different, but equivalent, views of the document:

$$Pair[X] \;\widehat{=}\; [\; Left, Right : X \;]$$

We use the symbol "Δ" throughout the specification to represent the conjunction of a before-state (undashed) and an after-state (dashed):

$$\Delta Pair[X] \;\widehat{=}\; Pair[X] \;\wedge\; Pair'[X]$$

and the symbol "Ξ" to represent a no change Δ-state:

$$\Xi Pair[X] \;\widehat{=}\; [\; \Delta Pair[X] \mid Pair[X] = Pair'[X] \;]$$

Finally, we use "Ξ_{Cont}" to represent a before- and after-document whose content hasn't changed (i.e. it allows for a change of current position):

$$\Xi_{Cont} Pair[X] \;\widehat{=}\; [\; \Delta Pair[X] \mid Left \frown Right = Left' \frown Right' \;]$$

1.1.1 The Generic Move And Delete Operations

We specify the basic operations move (left) and delete (left): the former has the pre-condition that *Left* must not be empty, its last element being moved to the beginning of *Right*:

$$\begin{array}{|l}
\textit{Mv}[X] \\
\hline
\Xi_{Cont} Pair \\
\hline
Left \neq \langle\, \rangle \\
Left' = \mathsf{front}\ Left \\
\hline
\end{array}$$

and the latter has the same pre-condition, but this operation discards the last element of *Left*, leaving *Right* unchanged:

$$\begin{array}{|l}
\textit{Del}[X] \\
\hline
\Delta Pair \\
\hline
Left \neq \langle\, \rangle \\
Left' = \mathsf{front}\ Left \\
Right' = Right \\
\hline
\end{array}$$

We also specify an operation which moves the current position to the end of the document, with pre-condition that that must not already be the current position:

$MvBot[X]$

$\Xi_{Cont}Pair$

$Right \neq <>$
$Right' = <>$

1.2 The Character, Word and Line Views Of The Document

A *Word* may be defined as a non-empty sequence of characters which consist entirely of whitespace (space characters) or non-whitespace (no spaces or newlines), or the unit sequence whose element is the newline character. A *Line* may be defined as a (possibly empty) sequence of characters not containing the newline character. We therefore introduce two special characters:

$sp, nl : Char \mid sp \neq nl$

and define:

$$WhiteSpace \mathrel{\widehat{=}} \{sp\}$$
$$NonWhiteSpace \mathrel{\widehat{=}} Char - \{sp, nl\}$$

to give:

$$Word \mathrel{\widehat{=}} seq_1\ WhiteSpace \cup seq_1\ NonWhiteSpace \cup <nl>$$
$$Line \mathrel{\widehat{=}} seq\ (Char - \{nl\})$$

We wish to define a total bijective function that converts a sequence of characters into a unique sequence of words; if we consider the sequence of characters:

$<t, h, e, sp, sp, c, a, t>$

we want the corresponding word sequence to be:

$<<t, h, e,>, <sp, sp>, <c, a, t,>>$

so that each member of the sequence satisfies the definition of *Word*, and the sequence flattens (through distributed concatenation) into the character sequence. However the following word sequence also satisfies those requirements:

$<<t, h, e>, <sp>, <sp>, <c, a>, <t>>$

In order to obtain the desired word view of the document, therefore, we must not allow two *Whitespace* words or two *NonWhiteSpace* words to be adjacent in that representation. Clearly, this requirement will ensure a unique word sequence for each sequence of characters. We define:

$$
\begin{array}{|l}
DocWordSeq : \mathbb{P}\ (seq\ Word) \\
\hline
W \in DocWordSeq\ \Leftrightarrow \\
\quad \forall\ w1, w2 : Word\ ;\ w1, w2 : Word\ \mid\ \langle w1, w2 \rangle\ \mathsf{infix}\ W\ \bullet \\
\qquad\qquad \mathsf{rng}\ (w1) = WhiteSpace\ \Rightarrow\ \mathsf{rng}\ (w2) \neq WhiteSpace \\
\qquad\qquad \mathsf{rng}\ (w1) = NonWhiteSpace\ \Rightarrow\ \mathsf{rng}\ (w2) \neq NonWhiteSpace
\end{array}
$$

where:

$$
\begin{array}{|l}
\_\ \mathsf{infix}\ \_ : seq\ X\ \times\ seq\ X \longrightarrow \mathbb{B} \\
\hline
X1\ \mathsf{infix}\ X\ \Leftrightarrow\ \exists\ X2, X3 : seq\ X\ \bullet\ X2 \frown X1 \frown X3 = X
\end{array}
$$

As a direct consequence of this definition, if we have two *DocWordSequences* such that their distributed concatenation is equal, the sequences themselves are equal:

Corollary 3 : 1.2a

$W1, W2 : DocWordSeq \mid\ \frown / W1 = \frown / W2$

$\vdash$

$W1 = W2$

■

Since by definition of " $\frown /$ ", if two sequences are equal, so is their distributed concatenation, we now define the required one-to-one function, FW (FlattenWords):

$$
\begin{array}{|l}
\mathsf{FW} : DocWordSeq \rightarrowtail\!\!\twoheadrightarrow seq\ Char \\
\hline
\mathsf{FW}\ W = \frown / W
\end{array}
$$

The sequence of words corresponding to the empty sequence of characters is the empty sequence of words:

Lemma 3 : 1.2b

$C = \langle\rangle$

$\vdash$

$\mathsf{FW}^{-1}\ C = \langle\rangle$

■

Proof

Follows directly from the definition of FW, since $\frown/(\langle\rangle) = \langle\rangle$

■

We also wish to define an analogous function that transforms the line view of the document into the character view. For example, we want the sequence of characters:

$$\langle c, a, n, a, r, y \rangle$$

to correspond to the line sequence:

$$\langle\langle c, a, n, a, r, y \rangle\rangle$$

and the character sequence:

$$\langle t, h, e, nl, c, a, n, a, r, y, nl, nl, a, t, e \rangle$$

to correspond to the line sequence:

$$\langle\langle t, h, e \rangle, \langle nl \rangle, \langle c, a, n, a, r, y \rangle, \langle nl \rangle, \langle nl \rangle, \langle a, t, e \rangle\rangle$$

so that the line sequence flattens (through distributed concatenation) to the character sequence. In order to ensure that each line view is unique, we define a *DocLineSeq*, analogous to *DocWordseq*, requiring that two non-newline words may not be adjacent in the representation:

$$DocLineSeq : \mathbf{P}\ (seq\ Line)$$

$$\forall\ L : DocLineSeq;\ l1, l2 : line \mid \langle l1, l2 \rangle \text{ infix } L \bullet$$
$$l1 \neq \langle nl \rangle \Rightarrow l2 = \langle nl \rangle$$

As a direct consequence of this definition, if we have two *DocLineSeq*uences such that their distributed concatenation is equal, the sequences themselves are equal:

Corollary 3 : 1.2c

$$L1, L2 : DocLineSeq \mid \frown/L1 = \frown/L2$$
$$\vdash$$
$$L1 = L2$$

■

Since by definition of " $\frown/$ ", if two sequences are equal, so is their distributed concatenation, we now define the required one-to-one function, FL (FlattenLines):

$$\mathsf{FL} : DocLineSeq \twoheadrightarrowtail seq\ Char$$

$$\mathsf{FL}\ L = \ ^\frown / L$$

The sequence of lines corresponding to the empty sequence of characters is the empty sequence of lines:

Lemma 3 : 1.2d

$$C = \langle \rangle$$

$$\vdash$$

$$\mathsf{FL}^{-1}\ C = \langle \rangle$$

■

Proof

Similar to Lemma 3:1.2b.

■

1.2.1 The Instantiated Move And Delete Operations

We now instantiate the formal parameter X with the sets of actual generic parameters *seq Char*, *DocWordSeq* and *DocLineSeq* to give schemas that specify movement and deletion by a character, word or line:

$$Mv_{Char} \ \widehat{=}\ Mv[seq\ Char]$$
$$Mv_{Word} \ \widehat{=}\ Mv[DocWordSeq]$$
$$Mv_{Line} \ \widehat{=}\ Mv[DocLineseq]$$

$$Del_{Char} \ \widehat{=}\ Del[seq\ Char]$$
$$Del_{Word} \ \widehat{=}\ Del[DocWordSeq]$$
$$Del_{Line} \ \widehat{=}\ Del[DocLineseq]$$

$$MvBot_{Char} \ \widehat{=}\ MvBot[seq\ Char]$$

Expanding the schemas for Mv_{Line} and Del_{Word}, for example, we have:

Mv_{Line}

$$Left, Right, Left', Right' : DocLineSeq$$

$$Left \frown Right = Left' \frown Right'$$
$$Left \neq \langle \rangle$$
$$Left' = \mathsf{front}\ Left$$

Del_{Word}

$Left, Right, Left', Right' : DocWordSeq$

$Left \neq \langle\rangle$
$Left' = \mathsf{front}\ Left$
$Right' = Right$

1.2.2 Some Results Concerning Characters, Words And Lines

Since move and delete operations involve the front and tail of sequences, we give the following results concerning those operators applied to the word and line view of the *Doc1* model in terms of the character view.

We first consider the tail of a word sequence. Suppose C is a non-empty sequence of characters, W the corresponding word sequence, W' the tail of W and C' the character sequence corresponding to W'. We consider three cases: the first element of W being a newline word, a whitespace word and a non-whitespace word.

For the first case, since the newline itself forms a word, C' is obtained from C by the removal of the newline, for example:

$$C = \langle nl, t, h, e, sp, c, a, t \rangle$$
$$W = \mathsf{FW}^{-1}\ C = \langle\langle nl \rangle, \langle t, h, e \rangle, \langle sp \rangle, \langle c, a, t, \rangle\rangle$$
$$W' = \mathsf{tail}\ W = \langle\langle t, h, e \rangle, \langle sp \rangle, \langle c, a, t, \rangle\rangle$$
$$C' = \mathsf{FW}\, W' = \langle t, h, e, sp, c, a, t \rangle$$

For the second case, C' will postfix C starting at the first non-space character, for example:

$$C = \langle sp, sp, t, h, e, sp, c, a, t \rangle$$
$$W = \mathsf{FW}^{-1}\ C = \langle\langle sp, sp \rangle, \langle t, h, e \rangle, \langle sp \rangle, \langle c, a, t, \rangle\rangle$$
$$W' = \mathsf{tail}\ W = \langle\langle t, h, e \rangle, \langle sp \rangle, \langle c, a, t, \rangle\rangle$$
$$C' = \mathsf{FW}\, W' = \langle t, h, e, sp, c, a, t \rangle$$

or C' will be empty if no such non-space character exists, for example:

$$C = \langle sp, sp, sp, sp \rangle$$
$$W = \mathsf{FW}^{-1}\ C = \langle\langle sp, sp, sp, sp \rangle\rangle$$
$$W' = \mathsf{tail}\ W = \langle\rangle$$
$$C' = \mathsf{FW}\, W' = \langle\rangle$$

Similarly for the last case, C' will postfix C starting at the first non-non-whitespace character (space or newline), or be empty if no such character exists.

We therefore have:

Lemma 3 : 1.2.2a

$$C \in seq_1\ Char$$
$$C' \in seq\ Char \mid C' = \mathsf{FW}(\mathsf{tail}\ \mathsf{FW}^{-1}\ C)$$

$\vdash$

$$C'\ \mathsf{postfix}\ C\ \wedge$$
$$\quad \mathsf{head}\ C = nl$$
$$\quad C' = \mathsf{tail}\ C$$
$$\vee$$
$$\quad C\ (\!|\ 1 \mathinner{..} \#C - \#C'\ |\!) \subseteq \{sp\}$$
$$\quad C' \neq \langle\rangle \Rightarrow C(\#C - \#C' + 1) \neq sp$$
$$\vee$$
$$\quad C\ (\!|\ 1 \mathinner{..} \#C - \#C'\ |\!) \cap \{sp, nl\} = \emptyset$$
$$\quad C' \neq \langle\rangle \Rightarrow C(\#C - \#C' + 1) \in \{sp, nl\}$$

■

Proof

1.	$\forall\ S : seq_1\ Char \bullet\ {}^\frown/(\mathsf{tail}\ S)\ \mathsf{postfix}\ {}^\frown/S$	propt. $^\frown/$
2.	${}^\frown/(\mathsf{tail}\ \mathsf{FW}^{-1}\ C)\ \mathsf{postfix}\ {}^\frown/(\mathsf{FW}^{-1}\ C)$	1.
3.	$\mathsf{FW}(\mathsf{tail}\ \mathsf{FW}^{-1}\ C)\ \mathsf{postfix}\ \mathsf{FW}(\mathsf{FW}^{-1}\ C)$	2., def.FW
4.	$C'\ \mathsf{postfix}\ C$	3., def.FW

The three disjunctions follow from the definition of *DocWordSeq*: the first followssince a newline word has length one, and the remaining two since, by definition, no two whitespace words and no two non-whitespace words can be adjacent in a *DocWordSeq*

■

In a similar way, we now consider the **front** of a line sequence. Suppose C is a non-empty seqnence of characters. L the corresponding line sequence. L' the **front** of L and C' the character sequence corresponding to L'. We consider two cases: the last element of L being a newline word, and being a non-newline word.

For the first case, C' is obtained from C by the removal of the newline character, for example:

$$C = \langle c, a, n, a, r, y, sp, a, t, e, nl \rangle$$
$$L = \mathsf{FL}^{-1}\ C = \langle\langle c, a, n, a, r, y, sp, a, t, e \rangle, \langle nl \rangle\rangle$$
$$L' = \mathsf{front}\ L = \langle\langle c, a, n, a, r, y, sp, a, t, e \rangle\rangle$$
$$C' = \mathsf{FL}\ L' = \langle c, a, n, a, r, y, sp, a, t, e \rangle$$

and for the second case, C' will prefix C up to the first newline character, for example:

$$C = < c, a, n, a, r, y, sp, a, t, e, nl, t, h, e >$$
$$L = \mathsf{FL}^{-1}\ C = << c, a, n, a, r, y, sp, a, t, e >, < nl >, < t, h, e >>$$
$$L' = \mathsf{front}\ L = << c, a, n, a, r, y, sp, a, t, e >, < nl >>$$
$$C' = \mathsf{FL}\ L' = < c, a, n, a, r, y, sp, a, t, e, nl >$$

or C' will be empty if no such non-space character exists, for example:

$$C = < c, a, n, a, r, y >$$
$$L = \mathsf{FL}^{-1}\ C = << c, a, n, a, r, y >>$$
$$L' = \mathsf{front}\ L = < >$$
$$C' = \mathsf{FL}\ L' = < >$$

Lemma 3 : 1.2.2b

$C \in seq_1\ Char$

$C' \in seq\ Char \mid C' = \mathsf{FL}(\mathsf{front}\ \mathsf{FL}^{-1}\ C)$

$\vdash$

C' prefix C $\wedge$

  last $C = nl$

  $C' =$ front C

$\vee$

  $nl \notin C ⦇ \# C' + 1 \ldots \# C ⦈$

  $C' \neq < > \;\Rightarrow\; C(\# C') \neq nl$

■

Proof

1. $\forall\ S : seq_1\ Char \;\bullet\; {}^\frown/(\mathsf{front}\ S)$ prefix ${}^\frown/S$ — propt. ${}^\frown/$
2. ${}^\frown/(\mathsf{tail}\ \mathsf{FW}^{-1}\ C)$ postfix ${}^\frown/(\mathsf{FW}^{-1}\ C)$ — 1.
3. $\mathsf{FW}(\mathsf{tail}\ \mathsf{FW}^{-1}\ C)$ postfix $\mathsf{FW}(\mathsf{FW}^{-1}\ C)$ — 2., def.FW
4. C' postfix C — 3., def.FW

The two disjunctions follow from the definition of *DocLineSeq*: the first follows since a newline word has length one, and second since, by definition, no two non-newline words can be adjacent in a *DocLineSeq*

■

1.3 The Doc1 State

We now wish to define a document state which incorporates the character, word and line views of the document, cementing the equivalence of the three views through a state invariant using the FW and FL functions.

This enables us to specify each operation on the appropriate view of the document, and since we specify on a "Δ" state which incorporates the state invariant, the other two equivalent views will be "automatically" updated.

We specify the three representations that are provided by $Pair_{Char}$, $Pair_{Word}$ and $Pair_{Line}$ which are schemas instantiating the generic set X of the $Pair$ schema of Section 2.1 with the sets $seq\ Char$, $DocWordSeq$ and $DocLineSeq$:

$$
\begin{aligned}
Pair_{Char} &\mathrel{\widehat=} Pair[seq\ Char] \\
Pair_{Word} &\mathrel{\widehat=} Pair[DocWordSeq] \\
Pair_{Line} &\mathrel{\widehat=} Pair[DocLineseq]
\end{aligned}
$$

and for the sake of brevity, we define:

$$
\begin{aligned}
Left_{Char} &\mathrel{\widehat=} Pair_{Char} \,.\, Left \\
Right_{Char} &\mathrel{\widehat=} Pair_{Char} \,.\, Right \\
Left_{Word} &\mathrel{\widehat=} Pair_{Word} \,.\, Left \\
Right_{Word} &\mathrel{\widehat=} Pair_{Word} \,.\, Right \\
Left_{Line} &\mathrel{\widehat=} Pair_{Line} \,.\, Left \\
Right_{Line} &\mathrel{\widehat=} Pair_{Line} \,.\, Right
\end{aligned}
$$

and incorporate an invariant relating the three views to give:

Doc1

$Pair_{Char}$
$Pair_{Word}$
$Pair_{Line}$

$Left_{Char} = \mathsf{FW}\ Left_{Word} = \mathsf{FL}\ Left_{Line}$
$Right_{Char} = \mathsf{FW}\ Right_{Word} = \mathsf{FL}\ Right_{Line}$

For non content-changing operations, we define:

$$\Xi_{Cont} Doc1 \mathrel{\widehat=} \Xi_{Cont} Pair[Char] \wedge \Delta Doc1$$

Initially the left and right character sequences are empty:

$$Initialize_{Doc1} \mathrel{\widehat=} [\ \Delta Doc1 \mid Left_{Char}' = Right_{Char}' = \langle\ \rangle\ ]$$

and we discharge PO 0, since Lemmas 3:1.2b and 3:1.2d give:

Lemma 3 : 1.3a

$$Initialize_{Doc1}$$

$\vdash$

$$Left_{Char}' = Right_{Char}' = \langle\,\rangle$$
$$Left_{Word}' = Right_{Word}' = \langle\,\rangle = \mathsf{FW}^{-1}(Left_{Char}') = \mathsf{FW}^{-1}(Right_{Char}')$$
$$Left_{Line}' = Right_{Line}' = \langle\,\rangle = \mathsf{FL}^{-1}(Left_{Char}') = \mathsf{FL}^{-1}(Right_{Char}')$$

■

We now promote the move and delete operations to the *Doc1* state:

$$\begin{array}{lcl} MvChar & \widehat{=} & Mv_{Char} \wedge \Delta Doc1 \\ MvWord & \widehat{=} & Mv_{Word} \wedge \Delta Doc1 \\ MvLine & \widehat{=} & Mv_{Line} \wedge \Delta Doc1 \\ \\ DelChar & \widehat{=} & Del_{Char} \wedge \Delta Doc1 \\ DelWord & \widehat{=} & Del_{Word} \wedge \Delta Doc1 \\ DelLine & \widehat{=} & Del_{Line} \wedge \Delta Doc1 \\ \\ MvBottom & \widehat{=} & MvBot_{Char} \wedge \Delta Doc1 \end{array}$$

We must discharge PO 1 for each of these operations: for example, for *DelWord* we have:

$$Left_{Char} = \mathsf{FW}\ Left_{Word} = \mathsf{FL}\ Left_{Line}$$
$$Right_{Char} = \mathsf{FW}\ Right_{Word} = \mathsf{FL}\ Right_{Line}$$
$$Left_{Word} \neq \langle\,\rangle$$

$\vdash$

$$\begin{array}{l} \exists\ Left_{Char}', Right_{Char}', Left_{Word}', Right_{Word}', Left_{Line}', Right_{Line}' \bullet \\ \qquad Left_{Word}' = \mathsf{front}\ (Left_{Word}) \\ \qquad Right_{Word}' = Right_{Word} \\ \qquad Left_{Char}' = \mathsf{FW}\ Left_{Word}' = \mathsf{FL}\ Left_{Line}' \\ \qquad Right_{Char}' = \mathsf{FW}\ Right_{Word}' = \mathsf{FL}\ Right_{Line}' \end{array}$$

which simplifies to:

$$Left_{Char} = \mathsf{FW}\ Left_{Word} = \mathsf{FL}\ Left_{Line}$$
$$Left_{Word} \neq \langle\,\rangle$$

$\vdash$

$$\begin{array}{l} \exists\ Left_{Char}', Left_{Line}' \bullet \\ \qquad Left_{Char}' = \mathsf{FW}\ \mathsf{front}\ (Left_{Word}) \\ \qquad Left_{Line}' = \mathsf{FL}^{-1}\ (Left_{Char}') \end{array}$$

Thus this and each of the other PO 1 proof obligations may be discharged since FW and FL are bijective.

1.3.1 The Insert Operation

We specify the operation to insert a character on the *Doc1* state, distinguishing the insertion of the tab character from a non-tab character. We introduce:

$tab : Char$

and first specify the operation to insert a non-tab character, x, at the end of the left character sequence, the right sequence remaining unchanged:

$$
\begin{array}{l}
\textit{InsNonTab} \\
\hline
\Delta Doc1 \\
x? : Char \\
\hline
x? \neq tab \\
Left_{Char}' = Left_{Char} \frown \langle x? \rangle \\
Right_{Char}' = Right_{Char} \\
\hline
\end{array}
$$

The tab character itself is not inserted into the document, but instead sufficient space characters are inserted at the end of the left character sequence to ensure that the current position is moved to the next (implementation- dependent) tabstop, with the right character sequence remaining unchanged:

$$
\begin{array}{l}
\textit{InsTab} \\
\hline
\Delta Doc1 \\
x? : Char \\
\hline
x? = tab \\
Left_{Char} \ \mathsf{prefix} \ Left_{Char}' \\
\mathsf{rng} \ (Left_{Char}' - Left_{Char}) = \{sp\} \\
Right_{Char}' = Right_{Char} \\
\hline
\end{array}
$$

to give, as the insert operation:

$InsChar \mathrel{\widehat=} InsNonTab \lor InsTab$

PO 1 may be discharged in the same way as in Section 1.3.

1.4 Introducing Direction

So far, the operations that we have defined have been "left" operations, and we now consider their "right" counterparts. We are able to derive the latter operations from the former by using of the schema

$$
\begin{array}{|l}
\hline
\textit{Mirror} \\
\Delta Doc1 \\
\hline
Left_{Char}' = \mathsf{rev}\ Right_{Char} \\
Right_{Char}' = \mathsf{rev}\ Left_{Char} \\
\hline
\end{array}
$$

and since $\mathsf{rev}\ ;\ \mathsf{rev}$ is the identity on sequences, we have:

$$Mirror\ ;\ Mirror \quad \widehat{=} \quad \Xi Doc1$$

If we apply *Mirror* followed by the left operation, followed by *Mirror* again we have achieved the corresponding right operation. To aid readability, we use the following syntactic shorthand:

$$
\begin{array}{lll}
\mathsf{Right}\ OP & \widehat{=} & Mirror\ ;\ OP\ ;\ Mirror \\
\mathsf{Left}\ OP & \widehat{=} & OP
\end{array}
$$

and apply to the move and delete operations defined in Section 2.2 to obtain:

$$
\begin{array}{lll}
RightMvChar & \widehat{=} & \mathsf{Right}\ MvChar \\
LeftMvChar & \widehat{=} & \mathsf{Left}\ MvChar \\
RightMvWord & \widehat{=} & \mathsf{Right}\ MvWord \\
LeftMvWord & \widehat{=} & \mathsf{Left}\ MvWord \\
RightMvLine & \widehat{=} & \mathsf{Right}\ MvLine \\
LeftMvLine & \widehat{=} & \mathsf{Left}\ MvLine \\
\\
RightDelChar & \widehat{=} & \mathsf{Right}\ DelChar \\
LeftDelChar & \widehat{=} & \mathsf{Left}\ DelChar \\
RightDelWord & \widehat{=} & \mathsf{Right}\ DelWord \\
LeftDelWord & \widehat{=} & \mathsf{Left}\ DelWord \\
RightDelLine & \widehat{=} & \mathsf{Right}\ DelLine \\
LeftDelLine & \widehat{=} & \mathsf{Left}\ DelLine \\
\\
MvToTop & \widehat{=} & \mathsf{Right}\ MvBottom \\
MvToBot & \widehat{=} & \mathsf{Left}\ MvBottom
\end{array}
$$

Partly expanding the schema *LeftMvLine*, for example, we obtain the operation initially requiring that the left line sequence does not comprise an empty line, with the left line sequence becoming equal to its front, and the right line sequence changing in such a way that the concatenation of the left and right line sequences does not change:

LeftMvLine

$\Delta Doc1$

$Left_{Line} \neq \langle\rangle$
$Left_{Line}' = \mathsf{front}\ Left_{Line}$
$Left_{Line}' \frown Right_{Line}' = Left_{Line} \frown Right_{Line}$

and similarly expanding the schema for *RightDelWord* we obtain an operation with the pre-condition that the right word sequence is non-empty, with the right word sequence becoming equal to its tail, and the left word sequence not changing:

RightDelWord

$\Delta Doc1$

$Right_{Word} \neq \langle\rangle$
$Right_{Word}' = \mathsf{tail}\ Right_{Word}$
$Left_{Word}' = Left_{Word}$

PO 1 for the "left" operations are discharged in Section 1.3; those for the "right" operations may be discharged in exactly the same way.

1.5 Error Messages

The move and delete commands of Section 1.4 have pre-conditions that either the right or left sequences must be non-empty. In order to make the operations total, we extend the operation domains by the inclusion of the report of an appropriate error message if an attempt is made to execute the command outside its domain: such error messages are assumed to belong to the set *Report*. We define:

$[Report]$

$Success \mathrel{\widehat{=}} [\ rep! : Report \mid rep! = \text{“OK”}\ ]$

$Doc1Unchanged \mathrel{\widehat{=}} [\ \Xi Doc1\ ;\ rep! : Report\ ]$

ErrorTopOfDoc

$Doc1Unchanged$

$Left_{Char} = \langle\rangle$
$rep! = \text{“At top of document”}$

ErrorBotOfDoc

Doc1Unchanged
$Right_{Char} = \langle\,\rangle$ $rep! =$ "At bottom of document"

Although we are not concerned with operational detail in this abstract specification, we must recognize that the editor will have finite capacity, and that the insert operation may not always be successful. We define:

ErrorFull

Doc1Unchanged
$rep! =$ "Editor full"

1.6 The Total Operations On The Doc1 State

We now have:

$RightMoveChar_{Doc1}$	$\widehat{=}$	$(RightMvChar \land Success)$	$\lor$	$ErrorBotOfDoc$
$LeftMoveChar_{Doc1}$	$\widehat{=}$	$(LeftMvChar \land Success)$	$\lor$	$ErrorTopOfDoc$
$RightMoveWord_{Doc1}$	$\widehat{=}$	$(RightMvWord \land Success)$	$\lor$	$ErrorBotOfDoc$
$LeftMoveWord_{Doc1}$	$\widehat{=}$	$(LeftMvWord \land Success)$	$\lor$	$ErrorTopOfDoc$
$RightMoveLine_{Doc1}$	$\widehat{=}$	$(RightMvLine \land Success)$	$\lor$	$ErrorBotOfDoc$
$LeftMoveLine_{Doc1}$	$\widehat{=}$	$(LeftMvLine \land Success)$	$\lor$	$ErrorTopOfDoc$
$RightDeleteChar_{Doc1}$	$\widehat{=}$	$(RightDelChar \land Success)$	$\lor$	$ErrorBotOfDoc$
$LeftDeleteChar_{Doc1}$	$\widehat{=}$	$(LeftDelChar \land Success)$	$\lor$	$ErrorTopOfDoc$
$RightDeleteWord_{Doc1}$	$\widehat{=}$	$(RightDelWord \land Success)$	$\lor$	$ErrorBotOfDoc$
$LeftDeleteWord_{Doc1}$	$\widehat{=}$	$(LeftDelWord \land Success)$	$\lor$	$ErrorTopOfDoc$
$RightDeleteLine_{Doc1}$	$\widehat{=}$	$(RightDelLine \land Success)$	$\lor$	$ErrorBotOfDoc$
$LeftDeleteLine_{Doc1}$	$\widehat{=}$	$(LeftDelLine \land Success)$	$\lor$	$ErrorTopOfDoc$
$InsertChar_{Doc1}$	$\widehat{=}$	$(InsChar \land Success)$	$\lor$	$ErrorFull$
$MoveToTop_{Doc1}$	$\widehat{=}$	$(MvToTop \land Success)$	$\lor$	$ErrorTopOfDoc$
$MoveToBot_{Doc1}$	$\widehat{=}$	$(MvToBot \land Success)$	$\lor$	$ErrorBotOfDoc$

Partly expanding the schema for $LeftMoveLine_{Doc1}$, for example, we obtain an operation which, if the left line sequence does not comprise an empty line performs *LeftMvLine* and issues an "OK" report, and if the left line sequence is an empty line does nothing except issue the "At top of document" report:

$$
\begin{array}{|l}
\textit{LeftMoveLine}_{Doc1} \\
\hline
\Delta Doc1 \\
rep! : Report \\
\hline
Left_{Line}' \frown Right_{Line}' = Left_{Line} \frown Right_{Line} \\
\quad Left_{Line} \neq \langle\rangle \\
\quad Left_{Line}' = \mathrm{front}\ Left_{Line} \\
\quad rep! = \text{"OK"} \\
\vee \\
\quad Left_{Line} = Left_{Line}' = \langle\rangle \\
\quad rep! = \text{"At top of document"} \\
\hline
\end{array}
$$

and similarly expanding the schema for $RightDeleteWord_{Doc1}$ defines an operation which, if the right word sequence is non-empty performs *RightDelWord* and issues a report of "OK", and if the right word sequence is empty does nothing except issue the report "At bottom of document":

$$
\begin{array}{|l}
\textit{RightDeleteWord}_{Doc1} \\
\hline
\Delta Doc1 \\
rep! : Report \\
\hline
Left_{Word}' = Left_{Word} \\
\quad Right_{Word} \neq \langle\rangle \\
\quad Right_{Word}' = \mathrm{tail}\ Right_{Word} \\
\quad rep! = \text{"OK"} \\
\vee \\
\quad Right_{Word} = Right_{Word}' = \langle\rangle \\
\quad rep! = \text{"At bottom of document"} \\
\hline
\end{array}
$$

Lemma 3:1.3a implies that the initialization operation will always succeed, and so we do not include a report message with that operation.

Since a *Success* schema does not alter the state, if we have discharged our invariant preservation obligation for some operation *OP*, then we have also discharged our obligation for the conjunction of that operation with a *Success* schema. An *Error* leaves the state unchanged, and so there is no associated proof obligation. Each of the above operations comprises a disjunction of two other operations, both of which preserve the state invariant, and thus each operation itself must preserve the state invariant.

2 Unbounded Display Of The Document

2.1 An Unbounded Display Model

In this section we specify a model which displays a document in full, and then (Section 8) use this unbounded model to develop a bounded display model incorporating a single movable window on to the document. We define the unbounded display in a manner orthogonal to that of *Doc1*.

An unbounded display may be uniquely characterised in many different ways. We choose a line model because this enables the display to be very naturally viewed as a sequence of lines placed one above the other and aligned at the left, the first at the top, the next immediately below, and so on.

For example, we want the four line display:

```
the canary
ate

the
```

to correspond to the sequence:

$$<< t, h, e, sp, c, a, n, a, r, y >, < a, t, e >, < >, < t, h, e >>$$

Note that each display line may be empty, but cannot contain a newline (unlike the line view incorporated in the *Doc1* model). The empty display will correspond to the unit sequence containing the empty display line.

We therefore define a display line as a sequence of characters not containing a newline:

$$DispLine \quad = \quad seq\ (Char - \{nl\})$$

and we characterize the display of an unbounded document by *UDLines*, a non-empty sequence of *DispLine*.

We model the screen cursor by a pair of natural numbers, with the top left hand position corresponding to $(1, 1)$, and we require the cursor to be inside the display of the document. If *UDCurX* and *UDCurY* represent the horizontal and vertical displacement from the top left hand corner of the screen, we therefore require that the latter cannot exceed the length of *UDLines*, and the former cannot be one more than the length of the $UDCurY^{th}$ line of *UDLines* (*UDCurX* attaining its greatest value when it appears immediately after the last character of that line).

For ease of reference, we also include *UDCurLine*, the $UDCurY^{th}$ line of *UDLines* (the line in which the cursor resides) in the unbounded display model:

UD

$UDLines : seq_1 \; DispLine$
$UDCurX, UDCurY : \mathbb{N}_1$
$UDCurLine : DispLine$

$UDCurY \leq \# UDLines$
$UDCurLine = UDLines \; UDCurY$
$UDCurX \leq \# UDCurLine + 1$

We note the following resnlt when the display contains a single empty line:

Lemma 3 : 2.1a

$UD \mid UDLines = \langle\langle\rangle\rangle$

$\vdash$

$UDCurY = 1$
$UDCurLine = \langle\rangle$
$UDCurX = 1$

■

Proof

$$UDCurY \in \mathbb{N}_1 \wedge UDCurY \leq \# \langle\langle\rangle\rangle \Rightarrow UDCurY = 1$$
$$UDCurLine = \langle\langle\rangle\rangle \; 1 = \langle\rangle$$
$$UDCurX \in \mathbb{N}_1 \wedge UDCurX \leq \# \langle\rangle + 1 \Rightarrow UDCurX = 1$$

■

2.2 The Doc2 State

The *Doc1* state is now extended to *Doc2* by incorporating its unbonnded display: We relate the two through their respective character seqnences, such that the flattened display line sequence is the concatenation of the left and right character views of *Doc1*.

To obtain the character sequence corresponding to a sequence of display lines, we define a fnnction FDL (FlattenDisplayLines) that inserts a newline character between adjacent display lines before flattening the sequence. Clearly, we have a fnnction which is bijective:

$$\text{FDL} : seq_1\ DispLine \twoheadrightarrow\!\!\!\!\rightarrowtail seq\ Char$$

$$\forall\ L1, L2 : seq_1\ DispLine\ ;\ l : DispLine \bullet$$
$$\text{FDL}\ (\langle l \rangle) = l$$
$$\text{FDL}\ (L1 \frown L2) = (\text{FDL}\ L1) \frown \langle nl \rangle \frown (\text{FDL}\ L2)$$

We establish three results relating to this definition.

Firstly, an empty sequence of characters corresponds to the display containing a single empty line:

Lemma 3 : 2.2a

$$C = \langle\rangle$$
$$L : seq_1\ DispLine \mid L = \text{FDL}^{-1}\ C$$
$$\vdash$$
$$L = \langle\langle\rangle\rangle$$

■

Proof

$$\text{FDL}^{-1}\ \langle\rangle = \langle\langle\rangle\rangle$$

■

Secondly, if L is a non-empty sequence of display lines, the number of newlines in the corresponding character sequence is one less than the length of L:

Lemma 3 : 2.2b

$$L : seq_1\ DispLine$$
$$\vdash$$
$$\#((\text{FDL}\ L) \rhd \{nl\}) = \#L - 1$$

■

Proof

We use induction on the length of the sequence L

Base : $L = \langle l \rangle$

1.	$\# L = 1$	Base
2.	$\mathsf{FDL}\ L = l$	Base, defn. FDL
3.	$\#((\mathsf{FDL}\ L) \rhd \{nl\}) = 0$	2., $l \in DispLine$
4.	$\#((\mathsf{FDL}\ L) \rhd \{nl\}) = \# L - 1$	3., 1.

Step : $L = L1 \frown \langle l \rangle$ where $L1 \neq \langle\rangle$

5.	$\#((\mathsf{FDL}\ L1) \rhd \{nl\}) = \# L1 - 1$	Step
6.	$\# L = \# L1 + 1$	Step
7.	$\mathsf{FDL}\ L = (\mathsf{FDL}\ L1) \frown \langle nl \rangle \frown l$	Step, defn. FDL
8.	$\#((\mathsf{FDL}\ L) \rhd \{nl\}) = \#((\mathsf{FDL}\ L1) \rhd \{nl\}) + 1$	7., $l \in DispLine$
9.	$\#((\mathsf{FDL}\ L) \rhd \{nl\}) = \# L - 1$	5., 6., 8.

■

Finally, the character sequence corresponding to a sequence of display lines of length at least two and whose last element is the empty line has a newline as its last element:

Lemma 3 : 2.2c

$$\begin{array}{l|l} L : seq_1\ DispLine & \# L \geq 2 \wedge \mathsf{last}\ L = \langle\rangle \\ C : seq\ Char & C = \mathsf{FDL}\ L \end{array}$$

$\vdash$

$\mathsf{last}\ C = nl$

■

Proof

1.	*Let* $L = L1 \frown \langle\langle\rangle\rangle$	
2.	$L1 \neq \langle\rangle$	1., $\# L \geq 2$
3.	$C = \mathsf{FDL}(L1 \frown \langle\langle\rangle\rangle)$	1.
4.	$C = (\mathsf{FDL}\ L1) \frown \langle nl \rangle \frown \langle\rangle$	2., 3., def. FDL
5.	$\mathsf{last}\ C = nl$	4., defs. $\frown$, $\langle\rangle$

■

We now define the *Doc2* state as the conjunction of the *Doc1* and *UD* models, requiring that the two character representations are the same. Then *UDCurLine* comprises the last of the display line sequence corresponding to $Left_{Char}$ concatenated with the *first* of the display line sequence corresponding to $Right_{Char}$; *UDCurY* equals the length of the display line sequence corresponding to $Left_{Char}$, and *UDCurX* equals the length of the last element of that sequence:

Doc2

$Doc1$
UD

$UDLines = \mathsf{FDL}^{-1}(Left_{Char} \frown Right_{Char})$
$UDCurLine = \mathsf{last}\ (\mathsf{FDL}^{-1}\ Left_{Char}) \frown \mathsf{first}\ (\mathsf{FDL}^{-1}\ Right_{Char})$
$UDCurY = \#(\mathsf{FDL}^{-1}\ Left_{Char})$
$UDCurX = \#(\mathsf{last}\ \mathsf{FDL}^{-1}\ Left_{Char}) + 1$

We note that the invariant of *Doc2* is consistent with that of *Doc1*, since the word and line views of the latter are not used in the specification of the former. We show that there is no conflict between the *Doc2* and *UD* invariants by assuming that the two character representations are the same (the first predicate of *Doc2*) and showing that the definitions of *UDCurLine*, *UDCurX* and *UDCurY* meet the *UD* invariant.

In order to show the first of these, we have, denoting $\mathsf{FDL}^{-1}\ Left_{Char}$ by L, $\mathsf{FDL}^{-1}\ Right_{Char}$ by R, *UDLines* by U, and *UDCurY* by n:

1. Let $L = L0 \frown \langle l \rangle$ and $\langle r \rangle \frown R0 = R$	defn. FDL
2. $\# L0 = n - 1$	1., $\# L = n$
3. $\mathsf{FDL}\ L \frown \mathsf{FDL}\ R =$ $\mathsf{FDL}\ L0 \frown \langle nl \rangle \frown \langle l \rangle \frown \langle r \rangle \frown \langle nl \rangle \frown \mathsf{FDL}\ R0$	2., defn. FDL
4. $\mathsf{FDL}\ L \frown \mathsf{FDL}\ R =$ $\mathsf{FDL}\ L0 \frown \langle nl \rangle \frown \langle l, r \rangle \frown \langle nl \rangle \frown \mathsf{FDL}\ R0$	3., defn. $\frown$
5. $\mathsf{FDL}\ L \frown \mathsf{FDL}\ R = \mathsf{FDL}\ (L0 \frown \langle l, r \rangle \frown R0)$	4., defn. FDL
6. $\mathsf{FDL}^{-1}\ (\mathsf{FDL}\ L \frown \mathsf{FDL}\ R) = L0 \frown \langle l, r \rangle \frown R0$	5., defn. FDL
7. $U = L0 \frown \langle l, r \rangle \frown R0$	6., *Doc2*, defn. FDL
8. $U\ n = \langle l, r \rangle$	2., 7., defn. $\frown$

The minimum value for *UDCurY* is given when $Left_{Char}$ is minimal in the document - when $Left_{Char}$ is empty:

$$UDCurY \;=\; \#\,\mathsf{FDL}^{-1}\ \langle\rangle \;=\; \#\langle\ \langle\rangle\ \rangle \;=\; 1$$

and the maximum value is given when $Left_{Char}$ is maximal - when $Right_{Char}$ is empty:

$$UDCurY \;=\; \#\,\mathsf{FDL}^{-1}\ (Left_{Char} \frown Right_{Char}) \;=\; \#\ UDLines$$

Finally, the minimum value for *UDCurX* will be given when the **last** of $\mathsf{FDL}^{-1}\ Left_{Char}$ is minimal in *UDCurLine* - when it is empty:

$$UDCurX \;=\; \#(\ \langle\rangle\ ) + 1 \;=\; 1$$

and its maximum value will occur when the **last** of $\mathsf{FDL}^{-1}\ Right_{Char}$ is maximal - when the **first** of $\mathsf{FDL}^{-1}\ Right_{Char}$ is empty:

$$
\begin{aligned}
UDCurX &= \#(\mathsf{last}\ (\mathsf{FDL}^{-1}\ Left_{Char}) \frown \mathsf{first}\ (\mathsf{FDL}^{-1}\ Right_{Char})) + 1 \\
&= \#\ UDCurLine + 1
\end{aligned}
$$

Hence there is no conflict between the invariants of *Doc2* and *UD*.

Initially:

$$Initialize_{Doc2} \ \widehat{=}\ Initialize_{Doc1} \ \wedge\ \Delta Doc2$$

and as a direct result of the definition of FDL and Lemmas 3:1.3a and 3:2.1b, we have:

Lemma 3 : 2.2d

$$Initialize_{Doc2}$$

$\vdash$

$$UDLines' = \langle \langle \rangle \rangle$$
$$UDCurLine' = \langle \rangle$$
$$(UDCurX', UDCurY') = (1, 1)$$

■

2.3 Promotion Of The Doc1 Operations To The Doc2 State

Each of the operations of the *Doc1* state is promoted to the *Doc2* state in the same way. In order to save unnecessary repetition in this and subsequent promotion processes, we use the following informal method: we define the set of names:

$$
\begin{aligned}
MoveOps \ \widehat{=}\ \{ & RightMoveChar, LeftMoveChar, \\
& RightMoveWord, LeftMoveWord, \\
& RightMoveLine, LeftMoveLine, \\
& MoveTopDoc, MoveBotDoc \}
\end{aligned}
$$

$$
\begin{aligned}
DeleteOps \ \widehat{=}\ \{ & RightDeleteChar, LeftDeleteChar, \\
& RightDeleteWord, LeftDeleteWord, \\
& RightDeleteLine, LeftDeleteLine \}
\end{aligned}
$$

$$InsertOp \ \widehat{=}\ \{ InsertChar \}$$

to give:

$$EditOps1 \ \widehat{=}\ MoveOps \cup DeleteOps \cup InsertOp$$

Then for each operation *OP* defined in Section 1.6 on the *Doc1* state and in the set *EditOps1*, we have:

$$\forall\ OP : EditOps1 \bullet OP_{Doc2} \ \widehat{=}\ OP_{Doc1} \ \wedge\ \Delta Doc2$$

The *Doc2* invariant ensures that each of the unbounded display components is uniquely defined in terms of the *Doc1* components, and satisfy the *UD* invariant, which means that the introduction of the *UD* variables does not result in additional proof obligations since they are redundant - they may be calculated from the *Doc1* and we have shown that they do not violate the state invariant.

3 Invariants On The Unbounded Display Model

3.1 The Doc3 State

It is impossible to identify trailing whitespace (space characters at the end of a line) on a terminal screen (except when the cursor is at the end of a line) and so we require that each document line (except the cursor line) has no trailing whitespace. We define a display line to be **visible** if and only if it is empty or its last character is not a space character:

$$\mathsf{visible}\ \_ : DispLine \longrightarrow \mathbb{B}$$

$$\mathsf{visible}\ l \Leftrightarrow (l \neq \langle\rangle \Rightarrow \mathsf{last}\ l \neq sp)$$

For the same reason, we require that the document has no trailing null lines, and we specify that a non-empty sequence of display lines is a **visibleseq** if and only if it is empty or its last element is non-null:

$$\mathsf{visibleseq}\ \_ : seq\ DispLine \longrightarrow \mathbb{B}$$

$$\mathsf{visibleseq}\ L \Leftrightarrow (L \neq \langle\langle\rangle\rangle \Rightarrow \mathsf{last}\ L \neq \langle\rangle)$$

Expressing the above definition in terms of the corresponding character sequence gives:

Lemma 3 : 3.1a

$L : seq_1\ DispLine$

$C : seq\ Char \mid \mathsf{FDL}\ L = C$

$\vdash$

$\mathsf{visibleseq}\ L \Leftrightarrow (C \neq \langle\rangle \Rightarrow \mathsf{last}\ C \neq nl)$

■

Proof

Follows from Lemmas 3:2.2a and 3:2.2c.

■

In the *Doc3* model we therefore require that every line except the cursor line is **visible**, that the cursor line itself is **visible** following the cursor position, and that the sequence of lines below the cursor line is a **visibleseq**. We define:

$$\begin{array}{l}
\textit{Doc3} \\
\hline
\textit{Doc2} \\
\hline
\forall\ i : 1 \,..\, \#\, \textit{UDLines} - \{\textit{UDCurY}\} \bullet \mathsf{visible}\ (\textit{UDLines}\ i) \\
\mathsf{visible}\ (\textit{UDCurLine}\ \mathsf{after}\ \textit{UDCurX} - 1) \\
\mathsf{visibleseq}\ (\textit{UDLines}\ \mathsf{after}\ \textit{UDCurY}) \\
\hline
\end{array}$$

where:

$$\begin{array}{l}
\hline
_\ \mathsf{after}\ _ : seq\ X \times \mathbb{N} \longrightarrow seq\ X \\
\hline
S\ \mathsf{after}\ N = \mathsf{suc}^{\mathsf{N}}\ ;\ (1 \,..\, \mathsf{N} \lhd \mathsf{S}) \\
\hline
\end{array}$$

Since the *Doc3* state introduces no new components, the initial state of *Doc3* is exactly that of *Doc2*:

$$\textit{Initialize}_{\textit{Doc3}} \;\widehat{=}\; \textit{Initialize}_{\textit{Doc2}}$$

We note that, after initialization, and using Lemma 3:2.2d, the first predicate of *Doc3* is trivially true (since $\#\, \textit{UDLines} = \textit{UDCurY} = 1$); the second predicate follows by definition of "**after**", noting that *UDCurLine* is initially empty and that the empty sequence is **visible**; the final predicate holds again by definition of "**after**" noting that, initially, *UDLines* **after** *UDCurY* is empty, and thus comprises a **visibleseq**. Thus we discharge PO 0.

3.2 Two Relations That Tidy The Display

We now consider the preservation of the two invariant requirements of the *Doc3* model.

We first consider the whitespace invariant, and define a relation between two display lines in which the first is visible and is the longest such that it is a prefix the second:

$$\begin{array}{l}
\hline
_\ \mathsf{visible_prefix}\ _ : \textit{DispLine} \times \textit{DispLine} \longrightarrow \mathbf{B} \\
\hline
L'\ \mathsf{visible_prefix}\ L \;\Leftrightarrow\; \mathsf{visible}\ L' \wedge \mathsf{rng}\ (L - L') \subseteq \{sp\} \\
\hline
\end{array}$$

Note that, since L and L' are sequences:

$$\text{rng } (L - L') \subseteq \{sp\} \quad \Leftrightarrow \quad \begin{array}{l} L' \text{ prefix } L \\ L \, (\!| \, \# L' + 1 \, .. \, \# L \, |\!) \subseteq \{sp\} \end{array}$$

and that visible prefix is reflexive for all visible lines:

Lemma 3 : 3.2a

visible L

$\vdash$

L visible_prefix L

■

Proof

1.	L prefix L	defn. prefix
2.	$L \, (\!| \, \# L + 1 \, .. \, \# L \, |\!) = \emptyset \subseteq \{sp\}$	defns. .., $\subseteq$
3.	L visible_prefix L	1..2.

■

We now consider the preservation of the null lines invariant of the *Doc3* model by defining an analogous relation on sequences of display lines, in which the first sequence is a visibleseq and is the longest such prefix of the second:

$$\_ \text{ visibleseq\_prefix } \_ : seq_1 \; DispLine \times seq_1 \; DispLine \longrightarrow \mathbf{B}$$

$$L' \text{ visibleseq\_prefix } L \Leftrightarrow \text{visibleseq } L' \wedge \text{rng } (L - L') \subseteq \{ <> \}$$

Relating this result to the corresponding character sequences gives:

Lemma 3 : 3.2b

$L, L' : seq_1 \; DispLine$

$C, C' : seq \; Char \mid C = \text{FDL } L \wedge C' = \text{FDL } L'$

$\vdash$

$$L' \text{ visibleseq\_prefix } L \quad \Leftrightarrow \quad \begin{array}{l} C' \text{ prefix } C \\ C' \neq <> \Rightarrow \text{last } C' \neq nl \\ C \, (\!| \, \# C' + 1 \, .. \, \# C \, |\!) \subseteq \{nl\} \end{array}$$

■

Proof

The result follows directly from the definition of FDL, and the results of Lemmas 3:2.2c and 3:3.1a.

■

We note that visibleseq_prefix is also reflexive for all visibleseq:

Lemma 3 : 3.2c

visibleseq L

$\vdash$

L visibleseq_prefix L

■

Proof

Similar to Lemma 3:3.2a.

■

3.3 Promotion of Doc2 Operations to the Doc3 State

When showing that an operation preserves the invariant of the state on which it is defined, we assume that the state invariant is true before the operation is invoked (PO 1). Thus the only display line that may contain violating trailing whitespace after an operation has been performed is the previous cursor line (which may, of course, not have changed).

We therefore define an operation that removes violating trailing whitespace from the previous cursor line, taking the previous value of the length of $Left_{Char}$ as the identifying input parameter, (since we will post-sequentially compose with this operation) leaving all other *Doc2* components unchanged.

If the cursor line has not changed, there must be no trailing whitespace to the right of the new cursor position, and so the cursor line after $UDCurY - 1$ must become a visible prefix of itself, with the cursor line to the left of that position remaining unchanged, and if the cursor line has changed, the whole of the previous cursor line must become a visible prefix of itself:

RemTrailWS

$$\Delta UD$$
$$prevCP? : \mathbb{N}$$

$$UDCurX', UDCurY' = UDCurX, UDCurY$$
$$\{prevUDCurY\} \ndres UDLines' = \{prevUDCurY\} \ndres UDLines$$
$$\quad prevUDCurY = UDCurY$$
$$\quad UDCurLine' \text{ for } UDCurX - 1 = UDCurLine \text{ for } UDCurX - 1$$
$$\quad (UDCurLine' \text{ after } UDCurX - 1) \text{ visible\_prefix } (UDCurLine \text{ after } UDCurX - 1)$$
$$\lor$$
$$\quad prevUDCurY \neq UDCurY$$
$$\quad (UDLines'\ prevUDCurY) \text{ visible\_prefix } (UDLines\ prevUDCurY)$$
$$\text{where}$$
$$\quad prevUDCurY = \#(\mathsf{FDL}^{-1}((Left_{Char} \frown Right_{Char}) \text{ for } prevCP?))$$

The first two predicates ensure that all lines except the previous cursor line and the cursor position do not change. the first disjunction treats the case when the cursor line does not change, and the second deals with a change of cursor line. In both cases, since all lines except the previous cursor line cannot have had trailing whitespace, and since visible prefix is reflexive (Lemma 3:3.2c), the *Doc3* trailing whitespace invariant is met.

We now define an operation that removes trailing null lines from that part of the document following the new cursor line (once again leaving the cursor position unchanged):

RemTrailNL

$$\Delta UD$$

$$UDCurX', UDCurY' = UDCurX, UDCurY$$
$$UDLines' \text{ for } UDCurY = UDLines \text{ for } UDCurY$$
$$(UDLines' \text{ after } UDCurY) \text{ visibleseq\_prefix } (UDLines \text{ after } UDCurY)$$

The first two predicates ensure that the cursor position and all lines above and including the new cursor line do not change. the last predicate ensures no trailing null lines after the new cursor line, which meets the *Doc3* null lines requirement.

We promote the operations of Section 2.3, defined on the *Doc2* state, to the *Doc3* state by sequential composition with the first of the above operations (to remove trailing whitespace), followed by sequential composition with the second (to remove trailing null lines). We define an operation to identify the previous cursor position:

$$FlagPrevCursor \mathrel{\widehat=} [\, Left_{Char} : \mathrm{seq}\ Char\ ;\ prevCP! : \mathbb{N} \mid prevCP! = \#\, Left_{Char} \,]$$

For each operation *OP* identified with the set *EditOps1* (Section 2.3), we have:

$$\forall\ OP : EditOps1 \bullet$$
$$\quad OP_{Doc3} \mathrel{\widehat=} FlagPrevCursor\ ;\ OP_{Doc2}\ ;\ RemTrailWS\ ;\ RemTrailNL$$

We note that each *Rem* operation is total, and therefore may freely post-compose with both.

4 Cursor Movement

4.1 The QP State

The eight move operations defined on the *Doc3* state (left and right, by character, word and line, and to the top and bottom of the document) facilitate cursor movement around the document, but do not enable positions "outside" its unbounded display to be reached.

The unbounded display of the document defines a "quarter plane" - a plane bounded by the top and left hand edges of the document, but unbounded to the right and below - and we now consider operations to move the cursor around that quarter plane.

We specify a state comprising a coordinate position within the quarter plane, orthogonal to the model developed so far:

$$QP \;\widehat{=}\; [\; QPCurX, QPCurY : \mathbf{N}_1 \;]$$

We wish to define cursor movement up, down, left and right, and first define horizontal and vertical moves (when, respectively, *QPCurY* and *QPCurX* do not change); we define a vertical move with the parameter y so that we may subsequently use the definition to specify vertical movement by character or page:

$$QPHorizMove \;\widehat{=}\; [\; \Delta QP \mid QPCurY' = QPCurY \;]$$

$$QPVertMove \;\widehat{=}\; [\; \Delta QP \,;\, y : \mathbf{N}_1 \mid QPCurX' = QPCurX \;]$$

where the parameter y represents the number of characters to be moved vertically (since we wish to specify vertical movement by a page as well as a character). We define:

```
QPCurUp ______________________
| QPVertMove
|--------------------
| QPCurY - y > 0
| QPCurY' = QPCurY - y
|______________________________
```

```
QPCurDown ______________________
| QPVertMove
|--------------------
| QPCurY' = QPCurY + y
|______________________________
```

$$\begin{array}{l} \textit{QPCurLeft} \\ \hline \textit{QPHorizMove} \\ \hline QPCurX > 1 \\ QPCurX' = QPCurX - 1 \end{array}$$

$$\begin{array}{l} \textit{QPCurRight} \\ \hline \textit{QPHorizMove} \\ \hline QPCurX' = QPCurX + 1 \end{array}$$

noting that the pre-conditions are necessary for leftward and upward movement to ensure the preservation of the *QP* invariant.

To totalise the operations (by which we mean transform each such that it becomes total) we define the following error messages, the first when upward movement would cause *QPCurY* to become non-positive, and the second when the cursor is at the top left hand corner of the plane when any leftward move would similarly violate the invariant:

$$\begin{array}{l} \textit{ErrorAtTopPage} \\ \hline \Xi QP \\ y : \mathbb{N} \\ rep! : Report \\ \hline QPCurY - y \leq 0 \\ rep! = \text{“At top page of document”} \end{array}$$

$$\begin{array}{l} \textit{ErrorQPAtTop} \\ \hline \Xi QP \\ rep! : Report \\ \hline QPCurX = QPCurY = 1 \\ rep! = \text{“At top of document”} \end{array}$$

The remaining case to consider is when the cursor is at the left edge of the document but not in the top line. In this case, rather than the left move operation failing, and an appropriate error message being issued (which, we feel, would be frustrating for the user), we decrease *QPCurY* by one without specifying the value of *QPCurX*. This non-determinism enables the operation to mimic the left *UD* cursor move (which will thus be to the end of the previous line) when conjoined with the *Doc3* state (which, we feel, would be what the user would actually expect of the operation). We define:

$$\begin{array}{|l}
\hline
QPCurToPrevLine \\
\Delta QP \\
\hline
QPCurX = 1 \\
QPCurY > 1 \\
QPCurY' = QPCurY - 1 \\
\hline
\end{array}$$

We now have the total operations:

$$\begin{array}{lcl}
CursorUp & \hat{=} & \quad QPCurUp \wedge Success \\
 & & \vee \\
 & & \quad ErrorAtTopPage \\
CursorDown & \hat{=} & QPCurDown \wedge Success \\
CursorLeft & \hat{=} & \quad QPCurLeft \wedge Success \\
 & & \vee \\
 & & \quad QPCurToPrevLine \wedge Success \\
 & & \vee \\
 & & \quad ErrorQPTop \\
CursorRight & \hat{=} & QPCurRight \wedge Success
\end{array}$$

each of which clearly preserves the state invariant, and so we discharge **PO 1**.

We now consider cursor movement vertically by a page: it is desirable that the page height should be less than than that of the terminal screen (*WindowHeight*, introduced formally in Section 8), thereby ensuring that some information contained in the current display is "carried over" to the next display. We introduce:

$$PageHeight : \mathbb{N}_1 \quad | \quad PageHeight < WindowHeight$$

and now define:

$$\begin{array}{lcllll}
CursorUpChar_{QP} & \hat{=} & [\ QPCursorUp & | & y = 1\ ] \\
CursorUpPage_{QP} & \hat{=} & [\ QPCursorUp & | & y = PageHeight\ ] \\
CursorDownChar_{QP} & \hat{=} & [\ QPCursorDown & | & y = 1\ ] \\
CursorDownPage_{QP} & \hat{=} & [\ QPCursorDown & | & y = PageHeight\ ] \\
CursorLeftChar_{QP} & \hat{=} & QPCursorLeft \\
CursorRightChar_{QP} & \hat{=} & QPCursorRight
\end{array}$$

4.2 The Doc4 State

We combine the *Doc3* and *QP* states into the *Doc4* state by logically conjoining the two, and requiring that the two cursor positions coincide:

$$Doc4 \mathrel{\widehat=} [\ Doc3 \ \wedge\ QP \ |\ UDCurX = QPCurX \ \wedge\ UDCurY = QPCurY\ ]$$

We specify the initialize operation as:

$$Initialize_{Doc4} \mathrel{\widehat=} Initialize_{Doc3} \wedge \Delta Doc4$$

which means that, by definition of *Doc4*, the *QP* cursor must initially be the same as the of the *UD* cursor:

Lemma 3 : 4.2a

$$Initialize_{Doc4}$$
$$\vdash$$
$$QPCurX' = QPCurY' = 1$$

■

and we discharge PO 0.

4.2.1 Promotion Of The Doc3 Operations To The Doc4 State

Since the *Doc4* state requires that the *QP* and *UD* cursors agree, we specify a total operation to alter the value of the former to the latter:

EquateQPWithUD

QP'
$UDCurX, UDCurY : \mathbb{N}$

$QPCurX', QPCurY' = UDCurX, UDCurY$

We now define, for each *OP* defined on the *Doc3* state and associated with the set *EditOps1* of Section 2.3:

$$\forall\ OP : EditOps1 \bullet OP_{Doc4} \mathrel{\widehat=} (OP_{Doc3}\ ;\ EquateQPWithUD) \wedge \Delta Doc4$$

post-sequential composition with the *Equate* operation ensuring that each operation preserves the *Doc4* invariant.

4.2.2 Promotion Of The QP Operations To The Doc4 State

In order to preserve the *Doc4* invariant for each of the *QP* cursor movement operations, we must specify an operation to change the values of the *UD* cursor, but in so doing we may violate the invariant of the *Doc3* model. If the cursor position remains inside the unbounded display, the treatment of the operation is the same as that for a *Doc3* operation, but when the cursor position is moved outside, we must change the contents of the document, ensuring that the whitespace/null lines requirement is met.

As with the *UD* operations, the only line that may contain violating whitespace after the operation is the previous cursor line, and so for each *QP* operation, we post-sequentially compose with *RemTrailWS*. To ensure that there are no trailing null lines after the new cursor line we also post-sequentially compose with *RemTrailNL*. In both cases the cursor position does not change, and the considerations are exactly the same as those discussed in Section 3.3. we now consider the two cases when the cursor is moved outside the unbounded display.

We first consider the case when the *QP* cursor position is to the right of the unbounded display, when the cursor line position will not have changed. We may leave all lines unchanged except for this one (since we are, in effect, performing a left insert operation, which affects only the cursor line), and we choose to "pad" this line with whitespace (since this is what the user would naturally perceive on a terminal screen). Thus the previous cursor line (the $QPCurY^{th}$ line of *UDLines*) will be a **visible_prefix** of the new (the $QPCurY^{th}$ line of *UDLines'* - i.e. *UDCurLine'*). Note that since **visible_prefix** is reflexive (Lemma 3:3.2a) this relation will also hold when it is not necessary to change the contents of the line.

We now consider the *QP* cursor being moved below the bottom of the unbounded display. In this case we choose to pad the end of the document with empty lines (again fitting in with the users's perception of the display). Further, if the cursor is not at the left hand edge of the document, we pad the last of these lines with space characters. In all cases, all display lines in the (possibly empty) range $\# \mathit{UDLines} + 1$ to $\mathit{QPCurY} - 1$ will be empty, and the **rng** of the last line (which in this case will be the current line *UDCurLine'*) will be a subset of $\{sp\}$ (which also holds, of course, when that line is empty).

We define the operation to equate the *UD* with the *QP* cursor:

EquateUDWithQP

$$QP$$
$$\mathit{UDCurX}', \mathit{UDCurY}' : \mathbb{N}$$

$$\mathit{UDCurX}', \mathit{UDCurY}' = \mathit{QPCurX}, \mathit{QPCurY}$$

and the operation to pad the display with whitespace/newlines:

PadWSNL

ΔUD
$EquateUDWithQP$

$$QPCurY \leq \# UDLines$$
$$\{QPCurY\} \mathbin{⩤} UDLines' = \{QPCurY\} \mathbin{⩤} UDLines$$
$$UDLines\ QPCurY \ \textsf{visible\_prefix}\ UDCurLine'$$

$\vee$

$$QPCurY > \# UDLines$$
$$\# UDLines' = QPCurY$$
$$UDLines \ \textsf{prefix}\ UDLines'$$
$$\forall\ i : \# UDLines + 1 \mathrel{..} QPCurY - 1 \bullet UDLines'\ i = \langle\ \rangle$$
$$\textsf{rng}\ (UDCurLine') \subseteq \{sp\}$$

The first disjunction treats the case when the cursor does not move below the unbounded display: the second predicate ensures that no lines change except the cursor line, and the third allows whitespace padding of that line. The second disjunction deals with the cursor being moved below the unbounded display: the second predicate extends the number of display lines in the document to agree with $QPCurY$, the third predicate ensures that existing display lines do not change, the next ensures that all lines (except the last) appended to the display are empty, and the final predicate allows for whitespace padding of the last (cursor) line. The operation is total, and thus post-sequential composition with this operation will ensure that all newly added lines satisfy the *Doc3* invariant.

The *CursorLeftChar* operation is identical to $LeftMoveChar_{Doc3}$ (since the *QP* cursor will always remain inside the unbounded display):

$$CursorLeftChar_{Doc4} \ \widehat{=}\ LeftMoveChar_{Doc3} \wedge \Delta Doc4$$

We now promote the remaining *QP* cursor operations by defining the set of names:

$$QPCursorOps \ \widehat{=}\ \{\ CursorUpChar, CursorDownChar, CursorUpPage, CursorDownPage, CursorLeftChar, CursorRightChar\ \}$$

and have, recognizing that the editor's capacity may be exceeded:

$$\forall\ OP : QPCursorOps - \{CursorLeftChar\} \bullet$$
$$\begin{array}{lll} OP_{Doc4} & \widehat{=} & FlagPrevCursor\ ; \\ & & \quad SuccOP_{QP}\ ;\ PadWSNL\ ; \\ & & \quad RemTrailWS\ ;\ RemTrailNL \wedge \Delta Doc4 \\ & & \vee \\ & & \quad UnSuccOP_{QP} \wedge \Xi Doc4 \\ & & \vee \\ & & \quad ErrorFull \wedge \Xi Doc4 \end{array}$$

where

$$SuccOP_{QP} \;\widehat{=}\; [\ OP_{QP} \mid rep! = \text{“OK”}\ ]$$
$$UnSuccOP_{QP} \;\widehat{=}\; [\ OP_{QP} \mid rep! \neq \text{“OK”}\ ]$$

The last two disjunctions do not change the content of the operation, and for the first, the *Pad* operation ensures that the two cursor positions agree and, together with the two *Rem* operations, ensures that the *Doc3* invariant is maintained (and as explained in Section 3.3 we flag the cursor position before the start of each operation). Thus we discharge PO 1.

5 Text Manipulation Operations

5.1 The Doc5 State And Marked Text

It is sometimes necessary to identify a portion of text in order that it may either be removed from the document (and, possibly, replaced elsewhere) or lifted (to be copied elsewhere). Such text is referred to as “marked”. The operation to set the mark identifies a particular character position in the document and marked text is that lying between the mark and the cursor; hence marked text can lie above or below the cursor.

In the former case we define *MarkSeq* to be the sequence of characters starting at the top of the document and finishing at this marked position, and thus marked text, which we define as *MarkedSeq*, will be the sequence of characters lying between the marked and current positions. In the latter case *MarkSeq* will start at the marked position and end at the bottom of the document and *MarkedSeq* will lie between the current and marked positions. If the mark is not set, we define both *MarkSeq* and *MarkedSeq* to be empty. Thus when the mark is set moving the cursor increases or reduces the amount of text that is marked.

We extend the document state to incorporate marked text:

MarkedText

$MarkSeq, MarkedSeq : seq\ Char$
$Pair_{Char}$

$$MarkSeq = MarkedSeq = \langle\,\rangle$$
$\vee$
$$MarkSeq \frown MarkedSeq = Left_{Char}$$
$\vee$
$$MarkedSeq \frown MarkSeq = Right_{Char}$$

$$Doc5 \;\hat{=}\; Doc4 \;\wedge\; MarkedText$$

Initially the document does not contain marked text:

$$Initialize_{Doc5} \;\hat{=}\; [\; Initialize_{Doc4} \;\wedge\; \Delta Doc5 \;\mid\; MarkSeq = MarkedSeq = <> \;]$$

and thus we discharge PO 0 since the first predicate of *MarkedText* is satisfied (since $Initialize_{Doc4}$ ensures that $Left_{Char}$ is empty).

The Operation To Set The Mark

We define the operation to set the mark at the current cursor position:

```
 SetMk ______________________________
| MarkedText'
| Pair_Char
|-----------------
| MarkSeq' = Left_Char
| MarkedSeq' = < >
|______________________________________
```

which leaves the *Doc4* components of *Doc5* unchanged:

$$SetMark_{Doc5} \;\hat{=}\; SetMk \;\wedge\; \Xi Doc4 \;\wedge\; \Delta Doc5 \;\wedge\; Success$$

5.1.1 Promotion Of The Doc4 Operations To The Doc5 State

It is desirable that cursor-changing operations should preserve the mark (set or unset) but necessary that content-changing operations reset the mark (for instance if part of the document is deleted then *Mark* could point beyond the end of the document). However, since cursor-changing operations may themselves change the content of the document (through the *RemTrailWS* or *RemTrailNL* operations of Section 3.3. or the *PadWSNL* operation of Section 4.2.2) we may not stipulate that the marked position does not change, because if whitespace/newlines are inserted/deleted at a point in the document above the marked position, preserving the mark will, in fact, move it relative to the rest of the document.

Thus our policy for promoting the *Doc4* operations is to require that all content-changing operations result in the mark being reset, but to adopt a non-deterministic approach for cursor-changing operations when the mark is already set, allowing the implementation policy to dictate when the mark should be reset for such operations; in the latter case, when the mark is not set, it will remain so.

We define:

$$ResetMark \;\widehat{=}\; [\; \Delta MarkedText \mid MarkSeq' = MarkedSeq' = <> \;]$$

and use *MoveOps*, *DeleteOps* and *InsertOp* (Section 2.3) and *QPCursorOps* (Section 4.2.2) to define:

$$\begin{aligned}
NonCursorOps &\;\widehat{=}\; DeleteOps \cup InsertOp \\
CursorOps &\;\widehat{=}\; MoveOps \cup QPCursorOps - NonCursorOps \\
CursorOps_NoMarkSet &\;\widehat{=}\; CursorOps \mid MarkSeq' = MarkedSeq' = <> \\
CursorOps_MarkSet &\;\widehat{=}\; CursorOps \mid MarkSeq' \frown MarkedSeq' \neq <>
\end{aligned}$$

to give:

$$\begin{aligned}
&\forall\; OP : NonCursorOps &\bullet\quad & OP_{Doc5} \;\widehat{=}\; OP_{Doc4} \wedge ResetMark \\
&\forall\; OP : CursorOps_NoMarkSet &\bullet\quad & OP_{Doc5} \;\widehat{=}\; OP_{Doc4} \wedge \Xi MarkedText \\
&\forall\; OP : CursorOps_MarkSet &\bullet\quad & OP_{Doc5} \;\widehat{=}\; OP_{Doc4} \wedge \Delta MarkedText
\end{aligned}$$

We note that there are no associated proof obligations associated with this promotion process since *Doc4* and *MarkedText* do not have variables in common, and there is no "cementing" invariant contained in *Doc5*.

5.2 The Doc6 State And The Lift, Cut And Paste Operations

Marked text may be placed into a paste buffer by a *Lift* or *Cut* operation (the former leaving the marked text in the document, the latter removing it) and subsequently copied from the buffer to a (new) cursor position by the *Paste* operation. Text in the buffer is not changed until a new *Lift* or *Cut*, and consequently several copies of the buffer may be made at different points in the document.

We enrich the document state as follows:

$$\begin{aligned}
PasteBuffer &\;\widehat{=}\; [\; PBuff : \text{seq } Char \;] \\
Doc6 &\;\widehat{=}\; Doc5 \wedge PasteBuffer
\end{aligned}$$

and initially we set the buffer to be the empty sequence:

$$Initialize_{Doc6} \;\widehat{=}\; [\; Initialize_{Doc5} \wedge \Delta Doc6 \mid PBuff' = <> \;]$$

The Lift Operation

We first define an operation in which non-empty marked text is copied to the paste buffer:

$$
\begin{array}{|l}
\hline
\textit{CopyMTextPBuff} \\
\Delta PasteBuffer \\
Left_{Char} \\
MarkedText \\
\hline
PBuff' = MarkedSeq \neq \langle\,\rangle \\
\hline
\end{array}
$$

In order to totalise the operation we define the error schema:

$$
\begin{array}{|l}
\hline
\textit{ErrorNoTextMarked} \\
\Xi Doc6 \\
rep! : Report \\
\hline
MarkedSeq = \langle\,\rangle \\
rep! = \text{"No text marked"} \\
\hline
\end{array}
$$

to give:

$$
\begin{array}{lcl}
Lift_{Doc6} & \widehat{=} & CopyMTextPBuff \land \Xi Doc5 \land \Delta Doc6 \land Success \\
& \lor & \\
& & ErrorNoTextMarked
\end{array}
$$

The Cut Operation

This operation is similar to *Lift*, except that the marked text is actually removed from the document, and thus the mark pointer must be reset. We define a total operation which removes marked text from the document and resets the mark:

$$
\begin{array}{|lcl}
\hline
\textit{RemMText} & & \\
\Delta Pair_{Char} & & \\
\Delta MarkedText & & \\
\hline
MarkSeq' = Markedseq' = \langle\,\rangle & & \\
MarkSeq \frown MarkedSeq = Left_{Char} & \Rightarrow & Left_{Char}' = MarkSeq \\
& & Right_{Char}' = Right_{Char} \\
MarkedSeq \frown MarkSeq = Left_{Char} & \Rightarrow & Right_{Char}' = MarkSeq \\
& & Left_{Char}' = Left_{Char} \\
\hline
\end{array}
$$

Since the content of the document is changed, we must ensure the preservation of the *Doc3* invariant, and have:

$$
\begin{array}{lcl}
Cut_{Doc6} & \widehat{=} & FlagPrevCursor\ ; \\
& & \quad CopyMTextPBuff\ ;\ RemMText\ ; \\
& & \quad RemTrailWS\ ;\ RemTrailNL \land Success \\
& \lor & \\
& & \quad ErrorNoTextMarked
\end{array}
$$

The Paste Operation

The *Paste* operation concatenates the (non-empty) paste buffer onto the end of $Left_{Char}$, with $Right_{Char}$ and the paste buffer being left unchanged: the pasted text becomes marked text and thus we set the mark to the original length of $Left_{Char}$:

```
Pst ____________________________________
| ΔPair_Char
| ΔMarkedText
| PasteBuffer
|_______________
| PBuff ≠ < >
| Left_Char' = Left_Char ⁀ PBuff
| Right_Char' = Right_Char
| MarkSeq' = Left_Char
| MarkedSeq' = < >
|_______________________________________

ErrorPBuffEmpty ________________________
| ΞDoc6
| rep! : Report
|_______________
| PBuff = < >
| rep! = "Paste buffer empty"
|_______________________________________
```

We acknowledge that the paste operation may cause the capacity of the editor to be exceeded, and again ensure that the *Doc3* invariant is met by post-sequential composition with the *Rem* operations of Section 3.3 to give:

$$
\begin{aligned}
Paste_{Doc6} \;\widehat{=}\;\; & FlagPrevCursor \; ; \\
& \quad Pst \; ; \; RemTrailWS \; ; \; RemTrailNL \;\wedge\; \Delta Doc6 \;\wedge\; Success \\
& \vee \\
& \quad \Xi Doc6 \;\wedge\; ErrorFull \\
& \vee \\
& \quad ErrorPBuffEmpty
\end{aligned}
$$

5.2.1 Promotion Of The Doc5 Operations To The Doc6 State

We use *CursorOps* and *NonCursorOps* (Section 5.1.1) to define the set of names:

$$EditOps2 \;\widehat{=}\; CursorOps \cup NonCursorOps \cup \{Mark\}$$

and stipulate that each operation *OP* in *EditOps2* leaves the paste buffer unchanged:

$$\forall\; OP : EditOps2 \bullet OP_{Doc6} \;\widehat{=}\; OP_{Doc5} \wedge \Xi PasteBuffer$$

We again note that there are no proof obligations associated with this promotion process, for the same reasons as those discussed in Section 5.1.1.

6 Quote Commands

6.1 The Quote Buffer

Unlike other commands, quoted commands are not necessarily single-key commands, and may require entry of text; we introduce a buffer, *QuoteBuffer*, into which such text may be directly typed and edited, which we specify as the concatenation of a pair of sequences of characters, thereby enabling character movement during editing:

QuoteBuffer

$$Left_{Quote}, Right_{Quote} : seq\ Char$$
$$QBuff : seq\ Char$$

$$QBuff = Left_{Quote} \frown Right_{Quote}$$

6.1.1 Operations To Edit The Quote Buffer

In general, text typed into *QuoteBuffer* will be short , and so we provide only the limited editing features of character movement, insertion and deletion. We define the insert operation, noting that we exclude the insertion of the tab character (since this will result in a varying number of spaces being introduced into the buffer) and the other quote edit operations in an analogous way to those defined on *Doc1*:

QteInsChar

$$\Delta QuoteBuffer$$
$$x? : Char$$

$$x? \neq tab$$
$$Left_{Quote}' = Left_{Quote} \frown \langle x? \rangle$$
$$Right_{Quote}' = Right_{Quote}$$

QteLeftDelChar

$$\Delta QuoteBuffer$$

$$Left_{Quote} \neq \langle\rangle$$
$$Left_{Quote}' = \text{front } Left_{Quote}$$
$$Right_{Quote}' = Right_{Quote}$$

QteRightDelChar

$$\Delta QuoteBuffer$$

$$Right_{Quote} \neq \langle\rangle$$
$$Left_{Quote}' = Left_{Quote}$$
$$Right_{Quote}' = \text{tail } Right_{Quote}$$

QteLeftMvChar

$$\Delta QuoteBuffer$$

$$Left_{Quote} \neq \langle\rangle$$
$$Left_{Quote}' \frown Right_{Quote}' = Left_{Quote} \frown Right_{Quote}$$
$$Left_{Quote}' = \text{front } Left_{Quote}$$

QteRightMvChar

$$\Delta QuoteBuffer$$

$$Right_{Quote} \neq \langle\rangle$$
$$Left_{Quote}' \frown Right_{Quote}' = Left_{Quote} \frown Right_{Quote}$$
$$Right_{Quote}' = \text{tail } Right_{Quote}$$

and the error messages:

$$ErrorQuote \mathrel{\widehat=} [\, \Xi QuoteBuffer \,;\, rep! : Report \,]$$

ErrorIllegalCharacter

$$ErrorQuote$$
$$x? : Char$$

$$x? = tab$$
$rep! =$ "Illegal quote character"

$$\begin{array}{|l} \hline \textit{ErrorQuoteFull} \\ \hline \textit{ErrorQuote} \\ \hline rep! = \text{“Quote buffer full”} \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \textit{ErrorTopQuote} \\ \hline \textit{ErrorQuote} \\ \hline Left_{Quote} = \langle\ \rangle \\ rep! = \text{“At top of quote buffer”} \\ \hline \end{array}$$

$$\begin{array}{|l} \hline \textit{ErrorBotQuote} \\ \hline \textit{ErrorQuote} \\ \hline Right_{Quote} = \langle\ \rangle \\ rep! = \text{“At bottom of quote buffer”} \\ \hline \end{array}$$

to give:

$$\begin{array}{lcl} InsertChar_{Quote} & \widehat{=} & (QteInsChar \land Success) \lor ErrorQuoteFull \lor \\ & & \qquad ErrorIllegalCharacter \\ LeftDeleteChar_{Quote} & \widehat{=} & (QteLeftDelChar \land Success) \lor ErrorTopQuote \\ RightDeleteChar_{Quote} & \widehat{=} & (QteRightDelChar \land Success) \lor ErrorBotQuote \\ LeftMoveChar_{Quote} & \widehat{=} & (QteLeftMvChar \land Success) \lor ErrorTopQuote \\ RightMoveChar_{Quote} & \widehat{=} & (QteRightMvChar \land Success) \lor ErrorBotQuote \end{array}$$

To discharge PO 2, we note that each operation is a disjuuction, only the first of which changes the buffer. Each of the first disjunctions of the first three operations explicitly set $Left_{Quote}'$ and $Right_{Quote}'$; for the first disjunction of the left move operation, we note that:

$$\begin{array}{l} \quad Left_{Quote}, Right_{Quote} : seq\ Char \land Left_{Quote} \neq \langle\ \rangle \\ \vdash \\ \quad \exists\ Left_{Quote}', Right_{Quote}' \bullet \\ \qquad\qquad Left_{Quote}', Right_{Quote}' : seq\ Char \\ \qquad\qquad Left_{Quote}' \frown Right_{Quote}' = Left_{Quote} \frown Right_{Quote} \\ \qquad\qquad Left_{Quote}' = \mathsf{front}\ Left_{Quote} \end{array}$$

simplifies to:

$$\begin{array}{l} \quad Left_{Quote}, Right_{Quote} : seq\ Char \land Left_{Quote} \neq \langle\ \rangle \\ \vdash \\ \quad \exists\ Right_{Quote}' \bullet \quad Right_{Quote}' : seq\ Char \\ \qquad\qquad\qquad\qquad Right_{Quote}' = \langle \mathsf{last}\ Left_{Quote} \rangle \frown Right_{Quote} \end{array}$$

which is *true*. We may treat the right operation in an analogous way, and since *QBuff* is redundant (it may be calculated from $Left_{Quote}$ and $Right_{Quote}$), we discharge PO 2.

6.2 Operating System I/O

Although we are not concerned with implementation detail in the abstract specification, we must make some high-level assumptions about the operating system under which the editor will run since several quote operations will require the facility to read from or write to files and, in order to simplify their specification, we assume the existence of three operating system operations, stating our assumptions of these operations in the following "specifications".

The first, *SysGetPtr*, returns a pointer to the computer's store, from which reading or writing is to commence, and takes a file name (a sequence of characters) and file mode ("*r*" for reading, "*w*" for re-writing - creating a new file, or emptying an existing file - and "*a*" for appending to the end of an existing file) as input parameters. If the operation is unsuccessful (for example the file might have read or write protection), *NullPtr* is returned. We assume the set of such pointers *Ptr*:

$[Ptr]$

$NullPtr : Ptr$

SysGetPtr

$filename? : seq\ Char$
$filemode?, filemode! : \{<r>, <w>, <a>\}$
$Sysptr! : Ptr$

$filemode! = filemode?$

and define

$$SuccSysGetPtr \mathrel{\widehat=} [\ SysGetPtr \mid Sysptr! \neq NullPtr\ ]$$
$$UnSuccSysGetPtr \mathrel{\widehat=} [\ SysGetPtr \mid Sysptr! = NullPtr\ ]$$

The second operation, *SysWrite*, takes the pointer returned by *SysGetPtr* and *WriteSeq*, the sequence of characters to be written, and returns the boolean variable *NoWriteError* indicating whether or not the operation was successful.

Although we are not concerned with the operational detail of how the computers filestore is changed by an operation, we assume that the filestore is a mapping from *Ptr* to *Cont*

$[Cont]$
$Store \mathrel{\widehat=} [\ FStore : Ptr \rightarrow Cont\ ]$

and define a function that converts the contents of a stored file into a sequence of characters:

$$\mathsf{storedseq} : Cont \longrightarrow \mathrm{seq}\ Char$$

We now define the operation *SysWrite*: it has the input parameters *Sysptr*, *WriteSeq*? and *filemode*? and returns the flag *NoWriteError* indicating the success or otherwise of the operation. If the operation was unsuccessful, *FStore* will not change, otherwise if *filemode*? indicates a write, its *Sysptr* element will now be associated with *WriteSeq* (through the **storedseq** function), and if *filemode*? indicates an append, that element will now be associated with the concatenation of its previous association concatenated with *WriteSeq*:

SysWrite

$$\begin{array}{l}
\Delta Store \\
Sysptr? : Ptr \\
WriteSeq? : \mathrm{seq}\ Char \\
filemode? : \{\langle w \rangle, \langle a \rangle\} \\
NoWriteError! : \mathbf{B} \\
\hline
\neg\ NoWriteError! \ \Rightarrow\ FStore = FStore' \\
NoWriteError!\ \wedge\ filemode? = \langle w \rangle\ \Rightarrow \\
\qquad \{Sysptr?\} \lhd FStore' = \{Sysptr?\} \lhd FStore \\
\qquad \mathsf{storedseq}(FStore'\ Sysptr?) = WriteSeq? \\
NoWriteError!\ \wedge\ filemode? = \langle a \rangle\ \Rightarrow \\
Sysptr? \in \mathrm{dom}\ FStore \\
\qquad \{Sysptr?\} \lhd FStore' = \{Sysptr?\} \lhd FStore \\
\qquad \mathsf{storedseq}(FStore'\ Sysptr?) = \mathsf{storedseq}(FStore\ Sysptr?) \frown WriteSeq?
\end{array}$$

$$\begin{array}{lcl}
SuccSysWrite & \widehat{=} & [\ SysWrite \mid NoWriteError!\ ] \\
UnSuccSysWrite & \widehat{=} & [\ SysWrite \mid \neg\ NoWriteError!\ ]
\end{array}$$

The final operation that we assume, *SysRead*, is analogous to *SysWrite*, takes the parameter *filemode* (which must equal "r") and the pointer returned by *SysGetPtr*, returning the flag *NoReadError* indicating its success or otherwise, and in the former case, the sequence of characters *ReadSeq*, associated with *Sysptr* in *Store* (through **storedseq**); in all cases *Store* rermains unchanged:

SysRead

$$\begin{array}{l}
\Xi Store \\
Sysptr? : Ptr \\
ReadSeq! : \mathrm{seq}\ Char \\
filemode? : \{\langle r \rangle\} \\
NoReadError! : \mathbf{B} \\
\hline
\neg\ NoReadError!\ \Rightarrow\ ReadSeq! = \mathsf{storedseq}(FStore\ Sysptr?)
\end{array}$$

$$SuccSysRead \;\widehat{=}\; [\, SysRead \mid NoReadError! \,]$$
$$UnSuccSysRead \;\widehat{=}\; [\, SysRead \mid \neg\, NoReadError! \,]$$

To totalise the read and write operations, we define the error messages:

$$ErrorCannotOpenFile \;\widehat{=}\; [\, rep! : Report \mid rep! = \text{"Cannot open file"} \,]$$
$$ErrorWritingFile \;\widehat{=}\; [\, rep! : Report \mid rep! = \text{"Error writing file"} \,]$$
$$ErrorReadingFile \;\widehat{=}\; [\, rep! : Report \mid rep! = \text{"Error reading file"} \,]$$

to give:

$$\begin{array}{l} WriteToStore \;\widehat{=} \\ \qquad SuccSysGetPtr \gg SuccSysWrite \wedge Success \\ \quad \vee \\ \qquad SuccSysGetPtr \gg UnSuccSysWrite \wedge ErrorWritingFile \\ \quad \vee \\ \qquad UnSuccSysGetPtr \wedge ErrorCannotOpenFile \end{array}$$

$$SuccWriteToStore \;\widehat{=}\; [\, WriteToStore \mid rep! = \text{"OK"} \,]$$
$$UnSuccWriteToStore \;\widehat{=}\; [\, WriteToStore \mid rep! \neq \text{"OK"} \,]$$

We note that *WriteToStore* contains the input parameters *filename?*, *filemode?* and *WriteSeq?*, which will be provided by the quote operation. We recognize that text read from store will be appended to the document and so must allow for the possibility that the editor's capacity will be exceeded, and define:

$$ErrorReadFull \;\widehat{=}\; [\, rep! : Report \mid rep! = \text{"Editor full"} \,]$$

$$\begin{array}{l} ReadFromStore \;\widehat{=} \\ \qquad SuccSysGetPtr \gg SuccSysRead \wedge Success \\ \quad \vee \\ \qquad SuccSysGetPtr \gg UnSuccSysRead \wedge ErrorReadingFile \\ \quad \vee \\ \qquad UnSuccSysGetPtr \wedge ErrorCannotOpenFile \\ \quad \vee \\ \qquad SuccGetSysPtr \wedge ErrorReadFull \end{array}$$

$$SuccReadFromStore \;\widehat{=}\; [\, ReadFromStore \mid rep! = \text{"OK"} \,]$$
$$UnSuccReadFromStore \;\widehat{=}\; [\, ReadFromStore \mid rep! \neq \text{"OK"} \,]$$

We note that *ReadFromStore* contains the input parameters *filename?* and *filemode?* (which will be provided by the quote operation) and returns the component *ReadSeq!*.

6.3 The Doc7 State

In order to enable text entered at the keyboard to be directed either to the document or to the *QuoteBuffer* it is necessary to incorporate two states into the editor: a $State_{Doc}$ for normal document editing and a $State_{Quote}$ for *QuoteBuffer* editing. Further, some quoted commands also require the file name (a sequence of characters), and we therefore extend the document state as follows:

$$DocState \;\hat{=}\; [\; State : \{State_{Doc}, State_{Quote}\} \;]$$

$$DocName \;\hat{=}\; [\; Name : seq\ Char \;]$$

$$Doc7 \;\hat{=}\; Doc6 \wedge QuoteBuffer \wedge DocState \wedge DocName$$

Name is to be provided by the (operating system) command to start editing, and is assumed to be input to the editor through that command; *QBuff* is initially set to the empty sequence, with the editor in $State_{Doc}$, giving:

```
Initialize_Doc7 ___________________________
| Initialize_Doc6
| ΔDoc7
| filename? : seq Char
|__________________
| QBuff' = < >
| State' = State_Doc
| Name' = filename?
|___________________________________________
```

6.4 Quoted Operations

All operations (with the exception of the search/replace commands - see Section 7) are begun and terminated by pressing a particular key (the quote key), unlike the other operations specified so far which require the implementation to "bind" each one to a different key. When the document is in $State_{Doc}$ and the quote command (through the quote key) changes nothing except the state (which is changed to $State_{Quote}$) and the quote buffer (which is emptied ready to receive text):

```
Quote_StateDoc ____________________________
| ΔDoc7
| ΞDoc6
| rep! : Report
|__________________
| Name' = Name
| State = State_Doc
| State' = State_Quote
| QBuff' = < >
| rep! = "OK"
|___________________________________________
```

After the appropriate text has been entered into *QBuff*, the quote key is pressed again (now, of course, in $State_{Quote}$) and this acts as a request for the operation - dictated by the *QBuff* text - to be carried out ("request" since some of these commands will be concerned with the operating system, and may, for a variety of reasons not concerned with the editor, fail). We are not concerned with the explicit specification of such operating system operations (although we do make limited assumptions about the input and output relating to each, Section 6.2).

In order to save unnecessary repetition in the specification of the quote operations, we define two operations that perform a quote request, distinguishing between a document content change - *QuoteRequestContentChange* - and no content change - *QuoteRequestNoChange*.

We first define the operation *QuoteRequest* which is executed in $State_{Quote}$ and terminates in $State_{Doc}$ with the document name, paste and quote buffers unchanged, allowing only the content of the document and marked text to change, and providing the *filename* and *filemode* input for operating system i/o:

QuoteRequest

$$\begin{array}{|l}
\Delta Doc7 \\
\Xi DocName \\
\Xi PasteBuffer \\
\Xi QuoteBuffer \\
filename! : seq\ Char \\
filemode! : \{< r >, < w >, < a >\} \\
\hline
State = State_{Quote} \\
State' = State_{Doc}
\end{array}$$

If a quote (request) operation does change the content of the document, the mark is reset:

$$QuoteRequestContentChange \ \widehat{=} \ [\ QuoteRequest \mid MarkSeq' = MarkedSeq = < > \ ]$$

and we note that the operation defines all components of the *Doc7* state except the content of the document, and therefore when using it we ensure that the whitespace and null lines invariant of *Doc3* is maintained (using the *Rem* operations of Section 3.3) which therefore preserve the *Doc7* invariant.

We finally define a quote (request) operation that allows neither document content nor marked text to change:

$$QuoteRequestNoChange \ \widehat{=} \ QuoteRequest \wedge \Xi Doc5$$

and we note that this operation defines all components of the *Doc7* state, and so when using this operation we ensure that the *Doc7* invariant is preserved.

Each quote operation, except abort and escape, will be identified by its first letter, with optional arguments being separated by space characters; we require that the abort operation is entered in full to preclude possibly disastrous consequences, the escape operation being identified by the character "!". We introduce:

$a, b, i, o, q, r, s, t, w, ! : Char$

6.4.1 The Abort Command

This command returns control to the operating system without saving the contents of the document to backing store. We assume the set *SysOp* of operating system instructions and introduce:

$[SysOp]$

$SysReturnControl : SysOp$

noting that it has no associated input. We define the command for requesting an abort, which is always successful:

RequestAbort

QuoteRequestNoChange
$\langle a, b, o, r, t \rangle = QBuff$ $rep! = \text{"OK"}$

to give:

$$Abort_{Quote} \mathrel{\widehat=} RequestAbort \; ; \; SysReturnControl$$

6.4.2 The Save Command

The command writes the entire content of the document to store; we note that the content of the document might not have changed since it was loaded from store, or since it was last written, and so define the error message:

$$ErrorDocNotChanged \mathrel{\widehat=} [\, rep! : Report \mid rep! = \text{"Document not changed"} \,]$$

This operation specifies the sequence of characters to be writen, *WriteSeq*, as the entire document, provides *filemode* as "*w*" and *filename* as the document name (as input to the *WriteToStore* operation of Section 6.2), and does not change the content of the document:

RequestSave

QuoteRequestNoChange
$WriteSeq! : seq\ Char$

$\langle s \rangle = QBuff$
$filemode! = \langle w \rangle$
$WriteSeq! = Left_{Char} \frown Right_{Char}$
$filename! = Name$

and we have:

$$
\begin{aligned}
Save_{Quot} \mathrel{\widehat{=}}\ & \\
& RequestSave \gg WriteToStore \\
\lor \ & \\
& RequestSave \gg ErrorDocNotChanged
\end{aligned}
$$

Our comments of Section 6.4 indicate that, since we are using *QuoteRequestNoChange*, there are no proof obligations associated with the *Save* operation.

6.4.3 The Write And Append Commands

These commands write or append non-empty marked text to a named file, and do not change the document. We define a "proper" prefix relation in which S is a proper prefix of T if and only if S is a prefix of T but not equal to it:

$\_\ \mathsf{prefix}_1\ \_ : seq\ X \times seq\ X \rightarrow \mathbb{B}$

$S\ \mathsf{prefix}_1\ T \Leftrightarrow S\ \mathsf{prefix}\ T \land S \neq T$

and now define the operation to write marked text, which specifies *filemode* as "w", *WriteSeq* as *MarkedSeq* and provides *filename* as the contents of the quote buffer following the first two characters (since these characters indicate the quoted operation required) as input to *WriteToStore*:

RequestWriteMarkedText

QuoteRequestNoChange
$WriteSeq! : seq\ Char$

$\langle w, sp \rangle\ \mathsf{prefix}_1\ QBuff$
$filemode! = \langle w \rangle$
$filename! = QBuff\ \mathsf{after}\ 2$
$WriteSeq! = MarkedSeq$

and the analogous operation to append marked text, with *filemode* specified as "a":

RequestAppendMarkedText

$$
\begin{array}{l}
QuoteRequestNoChange \\
WriteSeq! : \text{seq } Char \\
\hline
< a, sp > \ \textsf{prefix}_1 \ QBuff \\
filemode! = < a > \\
filename! = QBuff \ \textsf{after} \ 2 \\
WriteSeq! = MarkedSeq
\end{array}
$$

We require that the marked text is non-empty, and use the *ErrorNoTextMarked* schema of Section 5.2 to give:

$$
\begin{array}{l}
Write_{Quote} \ \widehat{=} \\
\qquad [RequestWriteMarkedText \mid MarkedSeq \neq < >] \gg WriteToStore \\
\quad \lor \\
\qquad RequestWriteMarkedText \land ErrorNoTextMarked
\end{array}
$$

$$
\begin{array}{l}
Append_{Quote} \ \widehat{=} \\
\qquad [RequestAppendMarkedText \mid MarkedSeq \neq < >] \gg WriteToStore \\
\quad \lor \\
\qquad RequestAppendMarkedText \land ErrorNoTextMarked
\end{array}
$$

As in the previous section, the use of *QuoteRequestNoChange* ensures that we have no proof obligations.

6.4.4 The Quit Command

This command has no argument and first performs a save operation (if necessary - i.e. if the document's content has changed) and if successful issues an "OK" report, subsequently returning control to the operating system; if the save is necessary, but unsuccessful, editing continues with the document unchanged. We define:

RequestQuit

$$
\begin{array}{l}
QuoteRequestNoChange \\
WriteSeq! : \text{seq } Char \\
\hline
WriteSeq! = MarkedSeq \\
< q > = QBuff \\
filename! = Name \\
filemode! = < w > \\
WriteSeq! = Left_{Char} \frown Right_{Char}
\end{array}
$$

to give:

$$
\begin{aligned}
Quit_{Quote} \;\widehat{=}\; & \\
& RequestQuit \gg SuccWriteToStore \,;\, SysReturnControl \\
\vee & \\
& RequestQuit \gg ErrorDocNotChanged \,;\, SysReturnControl \\
\vee & \\
& RequestQuit \gg UnSuccWriteToStore
\end{aligned}
$$

6.4.5 The Input Command

This command inserts text from a uamed file to the current cursor position, and we specify a reqnest operation to provide *filemode* as "r" and *filename* as the contents of the quote buffer following the first two characters as input for *ReadFromStore* (Section 6.2):

RequestInput

$QuoteRequestContentChange$

$< i, sp > \text{ prefix}_1\ QBuff$
$filemode! = < r >$
$filename! = QBuff \text{ after } 2$

the text from a successful input is the concatenated on to the end of the left character sequence:

InputReadSeq

$\Delta DocI$
$ReadSeq? : seq\ Char$

$Left_{Char}' = Left_{Char} \frown ReadSeq?$
$Right_{Char}' = Right_{Char}$

We note that the file might not exist, that it might be of au unsuitable type (e.g. not a text file) or that the input may cause the capacity of the editor to be exceeded. and so define:

$$ErrorFileNotExist \;\widehat{=}\; [\ rep! : Report \mid rep! = \text{"File does not exist"}\ ]$$
$$ErrorUnsuitableFile \;\widehat{=}\; [\ rep! : Report \mid rep! = \text{"Unsuitable file"}\ ]$$

to give:

$$
\begin{aligned}
&Input_{Quote} \;\widehat{=}\\
&\quad FlagPrevCursor\;;\\
&\qquad\quad (RequestInput \gg SuccReadFromStore \gg InputReadSeq)\;;\\
&\qquad\quad RemTrailWS\;;\;RemTrailNL\\
&\qquad \vee\\
&\qquad\quad RequestInput \gg UnSuccReadFromStore\\
&\qquad \vee\\
&\qquad\quad RequestInput \wedge \Xi Doc5 \wedge ErrorFileNotExist\\
&\qquad \vee\\
&\qquad\quad RequestInput \wedge \Xi Doc5 \wedge UnsuitableFile\\
&\qquad \vee\\
&\qquad\quad RequestInput \wedge \Xi Doc5 \wedge ErrorFull
\end{aligned}
$$

Only the first disjunction will change the document. and in line the comments made in Section 6.4 regarding *QuoteRequestContentChange* we ensure that the *Doc7* invariant is maintained.

6.4.6 The Move To Line Number Command

This command moves the cursor to the beginning of the line number indicated by the *QBuff* text; if this number exceeds the number of lines in the document, the cursor is positioned at the beginning of the last line.

We assume the set of numbers *NumChar*, a subset of *Char*:

$$NumChar \subset Char$$

and introduce a total function that converts a sequence of *NumChar* into a natural number:

$$\mathsf{ConvNum} : seq\ NumChar \longrightarrow \mathbb{N}$$

The command is a cursor movement command and so all *Doc7* components (except the cursor) remain unchanged. We define:

$$
\begin{array}{|l}
\hline
RequestMvLineNumber\\
\hline
QuoteRequestContentChange\\
\hline
\mathsf{ran}\ QBuff \subseteq NumChar\\
DocCurX' = 1\\
DocCurY' = \min\ (\mathsf{ConvNum}\ QBuff, \#\ UDLines)\\
\hline
\end{array}
$$

to give, noting that we must preserve the *Doc3* invariant:

$MoveLineNumber_{Quote} \mathrel{\widehat=}$

$\quad (FlagPrevCursor \; ; \; RequestMvLineNumber \; ;$
$\quad RemTrailWS \; ; \; RemTrailNL) \land Success$

Similar comments to those made regarding the discharge of the proof obligation for *Input* also apply here.

6.4.7 The Escape Command

Communication with the operating system from within the editor may be achieved through a *Quote* operation: *QBuff* text commencing with the "!" character consitutes a request for such a command, the test for the command itself being the quote buffer text following that character. We introduce an operating system interpretive command, *commandseq*, which accepts a sequence of characters, and performs the appropriate action:

$SysInterpret : SysOp$

We define:

```
Escape.ProvideText ___________________________________
| QuoteRequestNoChange
| commandseq! : seq Char
|______________________
| # QBuff > 1
| < ! > prefix₁ QBuff
| commandseq! = tail QBuff
|______________________________________________________
```

```
ErrorNoCommandGiven __________________________________
| QuoteRequestNoChange
| rep! : Report
|______________________
| < ! > = QBuff
| rep! = "No command given"
|______________________________________________________
```

to give:

$Eseape_{Quote} \mathrel{\widehat=}$

$\quad Escape\_ProvideText \gg SysInterpret \land Success$
$\lor$
$\quad ErrorNoCommmandGiven$

Note that although the system command may not succeed, the *Doc7* operation itself is successful.

6.5 The Quote Command

We now express the effect of pressing the quote key, when the editor is in *Quote* state, as a disjunction of the above operations. However, when *QBuff* contains text other than that defined in the previous sections describing the *Quote* operations, an error is reported. We define:

$$ValidQBuffContent \mathrel{\widehat=} \{ \langle a, b, o, r, t \rangle, \langle q \rangle, \langle s \rangle, \langle ! \rangle \}$$

$$ValidQBuffPrefix \mathrel{\widehat=} \{ \langle a, sp \rangle, \langle i, sp \rangle, \langle w, sp \rangle, \langle ! \rangle \}$$

ValidQBuffText

QuoteBuffer
$\exists\ s : ValidQBuffContent \bullet s = QBuff$ $\vee$ $\exists\ s : ValidQBuffPrefix \bullet s \text{ prefix}_1\ QBuff$ $\vee$ $\mathrm{rng}\ QBuff \subseteq NumChar$

to give the error message:

ErrorQuoteError

QuoteRequestNoChange $\neg\ ValidQBuffText$ $rep! : Report$
$rep! =$ "Quote error"

We therefore have:

$$\begin{aligned} Quote_{StateQuote} \mathrel{\widehat=}\ & Abort_{Quote} \vee Save_{Quote} \vee Write_{Quote} \vee \\ & Append_{Quote} \vee Quit_{Quote} \vee Input_{Quote} \vee \\ & Escape_{Quote} \vee ErrorQuoteError \end{aligned}$$

and we now express the *Quote* operation as the disjunction of the operations specified on the *Doc* and *Quote* states:

$$Quote_{Doc7} \mathrel{\widehat=} Quote_{StateDoc} \vee Quote_{StateQuote}$$

Since we have demonstrated that each individual operation preserves the invariant on *Doc7*, the disjunction of those operations must do likewise.

6.6 Promotion Of Quote Buffer Edit Operations To The Doc7 State

We use the same names for the quote buffer edit operations as those we specified on *Doc1*; the former are promoted to *Doc7* by stipulating that they do not affect the *Doc6* components, and the latter by stipulating that they have no affect on the quote buffer, document state or document name. We define the set of names:

$$QBuffEditOps \;\widehat{=}\; \{\ InsertChar, LeftMoveChar, RightMoveChar, CursorLeftChar, CursorRightChar, LeftDeleteChar, RightDeleteChar\ \}$$

and:

$$PromoteToDoc7 \;\widehat{=}\; \Xi QuoteBuffer \wedge \Xi DocState \wedge \Xi DocName$$

to give:

$$\forall\ OP : QBuffEditOps \bullet$$
$$OP_{Doc7} \;\widehat{=}\; [\ OP_{Quote} \wedge \Xi Doc6 \mid State = State_{Quote}\ ] \vee [\ OP_{Doc6} \wedge PromoteToDoc7 \mid State = State_{Doc}\ ]$$

and, clearly, we have no proof obligations associated with this promotion.

6.7 Promotion Of Remaining Doc6 Operations To The Doc7 State

We note that the remaining *Doc6* operations may successfully be effected only in $State_{Doc}$, and they are promoted in the same way as those described above. We use *EditOps2* (Section 5.2.1) to define the set of names:

$$EditOps3 \;\widehat{=}\; EditOps2 \cup \{\ Lift, Cut, Paste, ExchMTextPBuff\ \}$$

to give:

$$NonQBuffEditOps \;\widehat{=}\; EditOps3 - QBuffEditOps$$

We define the error schema:

ErrorIllegalQBuffEditOp

$\Xi Doc7$

$State = State_{Quote}$
$rep! =$ "Illegal edit operation"

to give:

$$\forall\ OP : NonQBuffEditOps \bullet$$
$$OP_{Doc7} \ \widehat{=}\ \begin{array}{l}[\ OP_{Doc8} \wedge PromoteToDoc7 \mid State = State_{Doc}\ ] \\ \vee \\ ErrorIllegalQBuffEditOp\end{array}$$

and again we have no associated proof obligation.

7 The Search And Replace Operations

We now consider the operations to search for a specific string of characters, and (possibly) to replace that string with another specified string of characters. Since we wish all document changes to take place at the cursor position we do not allow "global" string replacement, and specify the replace operation as having the pre-condition that the cursor must be at the start of text matching that specified in the search operation.

7.1 The Doc8 State

We enrich the document state by providing two buffers:

$$SearchBuffer \ \widehat{=}\ [\ SBuff : seq\ Char\ ]$$
$$ReplaceBuffer \ \widehat{=}\ [\ RBuff : seq\ Char\ ]$$
$$Doc8 \ \widehat{=}\ Doc7 \wedge SearchBuffer \wedge ReplaceBuffer$$

and initially each buffer is set to the empty sequence:

$$Initialize_{Doc8} \ \widehat{=}\ [\ Initialize_{Doc7} \wedge \Delta Doc8 \mid SBuff' = RBuff' = \langle\rangle\ ]$$

As described in Section 6.3 pressing the quote key - *QuoteKey* - change states. Each time *QuoteKey* is pressed in $State_{Quote}$, the quote buffer is emptied ready to accept new text. Pressing the search key - *SearchKey* - will then have three effects: copying the contents of the quote buffer into the search buffer, carrying out a search operation for that text, and returning the editor to $State_{Doc}$. The replace operation performs a similar function except that text immediately following the cursor in the document must match that in *SBuff* (as it would immediately following a successful search operation) to enable a replace operation to start.

For example, if we wanted to search for the string "*foo*" and replace it with the string "*baz*", we would type the following at the keyboard (with the document initially in $State_{Doc}$):

QuoteKey f o o SearchKey QuoteKey b a z ReplaceKey

Notice that both *SearchKey* and *ReplaceKey* are pressed in $State_{Quote}$, and afterwards the search buffer contains "*foo*" and the replace buffer contains "*baz*". To repeat the above search/replace operation we would type:

SearchKey ReplaceKey

the difference being that now both keys are pressed in $State_{Quote}$ which means that the current contents of the search and replace buffers are nsed.

7.2 Regular Expressions

We wish to use a form of "regular expression" when searching for a string of characters and introduce:

$$RegExpression : \mathbf{P}\ (seq\ Char)$$

and define a relation which holds when a regular expression matches a prefix of a sequence of characters, and require that an expression cannot match by a sequence that is shorter in length:

$$\_\ \mathsf{regexpmatches}\ \_ : RegExpression \times seq\ Char \longrightarrow \mathbf{B}$$

$$e\ \mathsf{regexpmatches}\ s \;\Rightarrow\; \#e \geq s$$

Since we wish the document to be in a matched state when text in the search buffer matches text immediately following the cursor, we now define a relation between sequences of characters, the first of which may contain a bracket expression:

$$\_\ \mathsf{matches}\ \_ : seq\ Char \times seq\ Char \longrightarrow \mathbf{B}$$

$$\begin{aligned} &\forall\ e : RegExpression;\ s1, s2 \in seq\ Char \mid s1 \notin RegExpression \bullet \\ &\qquad e\ \mathsf{matches}\ s2 \Leftrightarrow e\ \mathsf{regexpmatches}\ s \\ &\qquad s1\ \mathsf{matches}\ s2 \Leftrightarrow s1\ \mathsf{prefix}\ s2 \end{aligned}$$

We note that a sequence cannot be matched by one that is shorter in length.

7.3 The Down Search Operation

Search operations are cursor-changing operations and will start only when the search bnffer (which does not change) is non-empty, and will terminate in $State_{Doc}$. We define:

$SearchOp$

$$\begin{array}{l} \Xi_{Cont} Doc1 \\ \Xi SearchBuffer \\ \Delta DocState \\ \hline SBuff \neq \langle\rangle \\ State' = State_{Doc} \end{array}$$

We need a means of determining the length of the matched string in the document, and so we define a partial function which takes two matched sequences (the first of which, of course, may be a regular expression) and returns the length of the match (i.e. the length of the matching prefix of the second sequence):

$$\begin{array}{l} \mathsf{matchedlength} : seq\ Char \times seq\ Char \rightarrowtail \mathbb{N} \\ \hline (s1, s2) \in \mathsf{dom\ matchedlength} \Leftrightarrow s1\ \mathsf{matches}\ s2 \end{array}$$

After a successful search, *SBuff* will match $Right_{Char}$ and this will be the first such available match - i.e. the text in *SBuff* must not match the content of the document from:

$$\# Left_{Char} + 2 \mathrel{..} \# Left_{Char}' + \mathsf{matchedlength}\ (SBuff, Right_{Char}) - 1$$

since the search will have started from the second element of $Right_{Char}$, and the match is with the first **matchedlength** *SBuff* elements of $Right_{Char}'$. We define a successful down search in $State_{Doc}$, which has the pre-condition that the length of $Right_{Char}$ must be at least that of the length of the search buffer:

$SuccDownSch_{Doc}$

$$\begin{array}{l} SearchOp \\ \hline State = State_{Doc} \\ \# Right_{Char} \geq \# SBuff \\ SBuff\ \mathsf{matches}\ Right_{Char}' \\ \neg(\exists\ S\ \mathsf{in}\ n \mathrel{..} m \upharpoonleft (Left_{Char} \frown Right_{Char}) \bullet SBuff\ \mathsf{matches}\ S) \\ \mathsf{where} \\ \quad n, m = (\# Left_{Char} + 2), (\# Left_{Char}' + \mathsf{matchedlength}\ (SBuff, Right_{Char}') - 1) \end{array}$$

The only difference between this and the corresponding operation in $State_{Quote}$ is that the quote buffer is first copied into the search buffer, with the operation terminating in $State_{Doc}$. We define:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{CopyQBuffSBuff}} \\
\hline
\Delta \mathit{SearchBuffer} \\
\Delta \mathit{DocState} \\
\mathit{QuoteBuffer} \\
\hline
\mathit{State} = \mathit{State}_{Quote} \\
\mathit{State}' = \mathit{State}_{Doc} \\
\mathit{SBuff}' = \mathit{QBuff} \\
\hline
\end{array}
$$

to give:

$$\mathit{SuccDownSch}_{Quote} \;\widehat{=}\; \mathit{CopyQBuffSBuff} \; ; \; \mathit{SuccDownSch}_{Doc}$$

An unsuccessful find in State_{Quote} occurs when the search buffer is not in the tail of Right_{Char} (since the search will start from its second element):

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{UnSuccDownSch}_{Doc}} \\
\hline
\mathit{SearchOp} \\
\Xi \mathit{Doc1} \\
\hline
\mathit{State} = \mathit{State}_{Doc} \\
\mathit{SBuff} \neq \langle\rangle \\
\neg (\exists\ S \textbf{ in } (\textsf{tail } \mathit{Right}_{Char}) \bullet \mathit{SBuff} \textsf{ matches } S) \\
\hline
\end{array}
$$

and we have the corresponding operation in *QuoteState*:

$$\mathit{UnSuccDownSch}_{Quote} \;\widehat{=}\; \mathit{CopyQBuffSBuff} \; ; \; \mathit{UnSuccDownSch}_{Doc}$$

To totalise the operation, we define the following report and error message:

$$\mathit{RepStringNotFound} \;\widehat{=}\; [\ rep! : \mathit{Report} \mid rep! = \text{“String not found”}\ ]$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\mathit{ErrorSBuffEmpty}_{Doc}} \\
\hline
\Xi \mathit{Doc8} \\
rep! : \mathit{Report} \\
\hline
\mathit{State} = \mathit{State}_{Doc} \\
\mathit{SBuff} = \langle\rangle \\
rep! = \text{“Search buffer empty”} \\
\hline
\end{array}
$$

$$\mathit{ErrorSBuffEmpty}_{Quote} \;\widehat{=}\; \mathit{CopyQBuffSBuff} \; ; \; \mathit{ErrorSBuffEmpty}_{Doc}$$

To give:

$$SuccDownSearch \mathrel{\widehat{=}} SuccDownSch_{Doc} \lor SuccDownSch_{Quote}$$
$$UnSuccDownSearch \mathrel{\widehat{=}} UnSuccDownSch_{Doc} \lor UnSuccDownSch_{Quote}$$
$$ErrorSBuffEmpty \mathrel{\widehat{=}} ErrorSBuffEmpty_{Doc} \lor ErrorSBuffEmpty_{Quote}$$

For each search operation, only the search buffer, document state and cursor position may change; however since a change of cursor position may result in a change of content (it may necessitate the removal of whitespace), we also unmark marked text in such cases. We define the promotion schema:

PromoteSearch

$\Delta Doc8$
$\Xi_{Cont} Doc1$
$\Delta MarkedText$
$\Xi PasteBuffer$
$\Xi QuoteBuffer$
$\Xi DocName$
$\Xi ReplaceBuffer$

$$PromoteSearchUnMark \mathrel{\widehat{=}} [\ PromoteSearch \mid MarkSeq = MarkedSeq' = \langle\rangle\ ]$$
$$PromoteSearchLeaveMark \mathrel{\widehat{=}} PromoteSearch \land \Xi MarkedText$$

to give:

$DownSearch_{Doc8} \mathrel{\widehat{=}}$
$\quad FlagPrevCursor$;
$\quad\quad SuccDownSearch$; $RemTrailWS$; $RemTrailNL \land$
$\quad\quad PromoteSearchUnMark \land Success$
$\quad \lor$
$\quad\quad UnSuccDownSearch \land PromoteSearchLeaveMark \land RepStringNotFound$
$\quad \lor$
$\quad\quad ErrorSBuffEmpty$

The last two disjunctions do not change the content of the document; for the first, we ensure the preservation of the *Doc3* invariant by post-sequential composition with the *Rem* operations of Section 3.3.

7.4 The Up Search Operation

We define searches up the document in an entirely analogous way. The difference occurs in the specification of the 'first match': the search buffer must not match the content of the document from:

$$\# Left_{Char}' + 2 \mathrel{..} \# Left_{Char} + \textsf{matchedlength}\ (SBuff, Right_{Char}) - 1$$

since the search will have started from the element corresponding to the penultimate character of the matched sequence of $Right_{Char}$ (or, if the search buffer is of unit length, from the last element of $Left_{Char}$ up to the second element of $Right_{Char}'$). We define:

$$\begin{array}{l}
\textit{SuccUpSch}_{Doc} \\
\hline
\textit{SearchOp} \\
\hline
State = State_{Doc} \\
\# Left_{Char} \geq \# SBuff \\
SBuff \text{ matches } Right_{Char}' \\
\neg (\exists\, S \text{ in } n \,..\, m \upharpoonleft (Left_{Char} \frown Right_{Char}) \bullet SBuff \text{ matches } S) \\
\text{where} \\
\quad n, m = (\# Left_{Char}' + 2), (\# Left_{Char} + \text{matchedlength } (SBuff, Right_{Char}') - 1) \\
\hline
\end{array}$$

$$SuccUpSch_{Quote} \;\widehat{=}\; CopyQBuffSBuff \;;\; SuccUpSch_{Doc}$$

An unsuccessful up find in $State_{Quote}$ occurs when the search buffer is not in the front of $Left_{Char}$:

$$\begin{array}{l}
\textit{UnSuccUpSch}_{Doc} \\
\hline
\textit{SearchOp} \\
\Xi Doc1 \\
\hline
State = State_{Doc} \\
SBuff \neq \langle\rangle \\
\neg (\exists\, S \text{ in } (\text{front } Left_{Char}) \bullet SBuff \text{ matches } S) \\
\hline
\end{array}$$

$$UnSuccUpSch_{Quote} \;\widehat{=}\; CopyQBuffSBuff \;;\; UnSuccUpSch_{Doc}$$

We may now specify the up search operation:

$$\begin{array}{lcl}
SuccUpSearch & \widehat{=} & SuccUpSch_{Doc} \lor SuccUpSch_{Quote} \\
UnSuccUpSearch & \widehat{=} & UnSuccUpSch_{Doc} \lor UnSuccUpSch_{Quote}
\end{array}$$

$$\begin{array}{l}
UpSearch_{Docs} \;\widehat{=} \\
\quad FlagPrevCursor \;; \\
\qquad SuccUpSearch \;;\; RemTrailWS \;;\; RemTrailNL \;\land \\
\qquad PromoteSearchUnMark \;\land\; Success \\
\quad \lor \\
\qquad UnSuccUpSearch \;\land\; PromoteSearchLeaveMark \;\land\; RepStringNotFound \\
\quad \lor \\
\qquad ErrorSBuffEmpty
\end{array}$$

The comments made regarding proof obligations for the promotion of the down search operation (Section 7.3) also apply here.

7.5 The Replace Operation

A *Replace* operation can only be successful when the document is in a "matched" state - i.e. when *SBuff* matches $Right_{Char}$, after which the document is left in $State_{Doc}$. The operation can be carried out in either $State_{Doc}$ or $State_{Quote}$ and we first consider the former: the text matched with that in the search buffer is first removed, the state remaining unchanged:

RemMatchedText

$$\begin{array}{l} \Delta Doc1 \\ \Xi DocState \\ SearchBuffer \\ \hline SBuff \text{ matches } Right_{Char} \\ Right_{Char}' = Right_{Char} \text{ after matchedlength } (SBuff, Right_{Char}) \\ Left_{Char}' = Left_{Char} \\ State = State_{Doc} \end{array}$$

and then the text in the replace buffer is concatenated on to the front of $Right_{Char}$:

InsRBuffText

$$\begin{array}{l} \Delta Doc1 \\ ReplaceBuffer \\ \hline Right_{Char}' = RBuff \frown Right_{Char} \\ Left_{Char}' = Left_{Char} \end{array}$$

and we have:

$$SuccRpl_{Doc} \mathrel{\widehat{=}} RemMatchedText \mathbin{;} InsRBuffText$$

The difference between this and the corresponding operation in $State_{Quote}$ is that the quote buffer is first copied into the replace buffer, after which the quote buffer is emptied. In a similar schema to *CopyQBuffSBuff* of Section 7.2, we define:

$$CopyQBuffRBuff \mathrel{\widehat{=}} CopyQBuffSBuff[ReplaceBuffer \setminus SearchBuffer]$$

to give:

$$SuccRpl_{Quote} \mathrel{\widehat{=}} CopyQBuffRBuff \mathbin{;} SuccRpl_{Doc}$$

An unsuccessful *Replace* operation in $State_{Doc}$ occurs when the document is not in a matched state, when neither the content nor the state change. We define:

$UnSuccRpl_{Doc}$

$\Xi Doc1$
$\Xi DocState$
$SearchBuffer$

$\neg\,(SBuff \text{ matches } Right_{Char})$
$State = State_{Doc}$

and have the corresponding operation in $State_{Quote}$:

$$UnSuccRpl_{Quote} \mathrel{\widehat=} CopyQBuffRBuff \; ; \; UnSuccRpl_{Doc}$$

We define the following error message:

$RepNoMatchSBuff$

$rep! : Report$

$rep! =$ "No match with search buffer"

Note that there is no replace schema analogous to the search schema *ErrorSBuffEmpty* since we want to make it possible to replace the found string with the null string - i.e. to enable a series of deletions to be made. For the replace operation, only the replace buffer, document state and document content may change (and in the latter case marked text must be unmarked), and we define:

$PromoteReplace$

$\Delta Doc8$
$\Xi_{Cont} Doc1$
$\Delta MarkedText$
$\Xi PasteBuffer$
$\Xi QuoteBuffer$
$\Xi DocName$
$\Xi SearchBuffer$

$$PromoteReplaceUnMark \mathrel{\widehat=} [\, PromoteReplace \mid MarkSeq = MarkedSeq' = \langle\,\rangle \,]$$
$$PromoteReplaceLeaveMark \mathrel{\widehat=} PromoteReplace \wedge \Xi MarkedText$$

and, recognizing that the *Replace* operation may exceed the editor's capacity, we define:

$ErrorReplaceFull_{Doc}$

$\Xi Doc8$

$State = State_{Doc}$
$rep! =$ "Editor full"

$$ErrorReplaceFull_{Quote} \quad \widehat{=} \quad CopyQBuffRBuff \ ; \ ErrorReplaceFull_{Doc}$$
$$ErrorReplaceFull \quad \widehat{=} \quad ErrorReplaceFull_{Doc} \ \vee \ ErrorReplaceFull_{Quote}$$

to give:

$$SuccReplace \quad \widehat{=} \quad SuccRpl_{Doc} \ \vee \ SuccRpl_{Quote}$$
$$UnSuccReplace \quad \widehat{=} \quad UnSuccRpl_{Doc} \ \vee \ UnSuccRpl_{Quote}$$

$$\begin{array}{l} Replace_{Doc8} \ \widehat{=} \\ \quad FlagPrevCursor \ ; \\ \qquad SuccReplace \ ; \ RemTrailWS \ ; \ RemTrailNL \ \wedge \\ \qquad PromoteReplaceUnMark \ \wedge \ Success \\ \quad \vee \\ \qquad UnSuccReplace \ \wedge \ PromoteReplaceLeaveMark \ \wedge \ RepNoMatchSBuff \\ \quad \vee \\ \qquad ErrorReplaceFull \end{array}$$

The comments made regarding proof obligations for the promotion of the down search operation (Section 7.3) also apply here.

7.6 Promotion Of The Doc7 Operations To The Doc8 State

We use *EditOps3* (Section 6.7) to define the set of names:

$$EditOps4 \ \widehat{=} \ EditOps3 \ \cup \ \{ \ Quote \ \}$$

We require that each operation in the set *EditOps4* does not change the search or replace buffers, to give:

$$\begin{array}{l} \forall \ OP : EditOps4 \ \bullet \\ \qquad OP_{Doc8} \ \widehat{=} \ OP_{Doc7} \ \wedge \ \Xi SearchBuffer \ \wedge \ \Xi ReplaceBuffer \end{array}$$

Clearly this promotion process does not incur proof obligations.

8 A Window On To the Display

In Section 2 we incorporate an unbounded display into the specification of the editor; in this section we specify a (movable) window on to that display. We first define a *Window* state, orthogonal to the *Doc* model.

8.1 The Window State

The window is assumed to be rectangular and of fixed width and height:

$$WinWidth, WinHeight : \mathbb{N}_1$$

We need a means of moving the window, and we introduce two values that represent its horizontal and vertical displacement from a fixed origin (so that the window always appears to the right of and below the origin):

$$WindowOffset \mathrel{\widehat=} [\ OffsetX, OffsetY : \mathbb{N}\ ]$$

The window contains a non-empty sequence of display lines (Section 2.1), the sequence having a maximum length of *WinHeight*, each line of the sequence being of maximum length *WinWidth*. The lines are displayed in the window one above the other, with the first line at the top of the window, the second immediately below it and so on, each vertically aligned with its left hand end flush against the left edge of the window.

We incorporate a cursor, a pair of positive natural numbers, such that the top left hand corner of the window corresponds with cursor position $(1, 1)$, the bottom right hand corner being $(WinWidth, WinHeight)$, and which we require to always be in the window:

WindowCursor

$$WinCurX, WinCurY : \mathbb{N}_1$$

$$1 \leq WinCurX \leq WinWidth$$
$$1 \leq WinCurY \leq WinHeight$$

and have:

Window

$$WindowLines : seq_1\ DispLine$$
$$WindowOffset$$
$$WindowCursor$$

$$\#\, WindowLines \leq WinHeight$$
$$\forall\ y : 1 \,..\, \#\, WinLines \bullet \#(WindowLines\ y) \leq WinWidth$$

8.2 The Doc9 State

We define the *Doc9* state by conjoining the *Doc8* and *Window* states. In order to obtain the sequence of window lines from the sequence of unbounded display lines we first mask ont that part of the unbonnded display lying to the left and right of the window position, to obtain the sequence *WinMaskLines*:

$$\forall\ y : 1 \mathinner{..} \#\ UDLines \bullet$$
$$WinMaskLines\ y = (UDLines\ y \ \textsf{after}\ OffsetX)\ \textsf{for}\ WinWidth$$

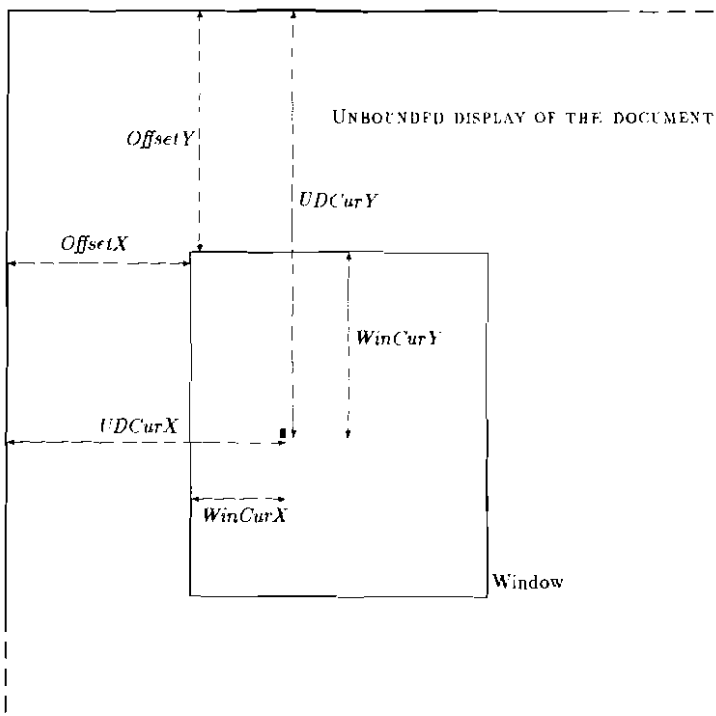
We note that the definition of for ensures that the length of each *WinMaskLine* does not exceed *WinWidth*.

We now mask out that part of *WimMaskLines* lying above and below the window:

$$WindowLines = (WinMaskLines \ \textsf{after}\ OffsetY)\ \textsf{for}\ WinHeight$$

and again, the definition of for ensures that the length of *WindowLines* does not exceed *WinHeight*.

The window cursor values will be the difference between the document cursor values and the offsets:



We now define:

$$
\begin{array}{|l}
\hline
\textit{Doc9} \\
\quad \textit{Doc8} \\
\quad \textit{Window} \\
\hline
\quad \textit{WinCurX}, \textit{WinCurY} = \textit{UDCurX} - \textit{OffsetX}, \textit{UDCurY} - \textit{OffsetY} \\
\quad \textit{WindowLines} = (\textit{WinMaskLines} \textsf{ after } \textit{OffsetY}) \textsf{ for } \textit{WinHeight} \\
\quad \text{where} \\
\qquad \# \textit{WinMaskLines} = \# \textit{UDLines} \\
\qquad \forall\ y : 1 \,..\, \# \textit{UDLines} \bullet \\
\qquad\qquad \textit{WinMaskLines}\ y = ((\textit{UDLines}\ y) \textsf{ after } \textit{OffsetX}) \textsf{ for } \textit{WinWidth} \\
\hline
\end{array}
$$

We note that the *WindowLines* sequence is redundant since it may be calculated from *UDLines*, the offsets and the window cursor; further, the offsets may be calculated from the window cursor, and vice-versa, using *Doc8* and the *UD* cursor. Hence *Doc8* and either the offsets or the window cursor uniquely define *Doc9*.

We show that the character at the (*WinCurY*, *WinCurX*) window position is the same as that at the (*UDCurX*, *UDCurY*) position of the unbounded display of the document:

Lemma 3 : 8.2a

$$
\begin{array}{l}
\quad \textit{Doc9} \\
\vdash \\
\quad (\textit{WindowLines}\ \textit{WinCurY})\ \textit{WinCurX} = (\textit{UDLines}\ \textit{UDCurY})\ \textit{UDCurX}
\end{array}
$$

■

Proof

1 $\quad WindowLines\ WinCurY =$
$((WinMaskLines \text{ after } OffsetY) \text{ for } WinHeight)(UDCurY - OffsetY)$ $\quad$ *Doc9*

2 $\quad 1 \leq WinCurY \leq WinHeight$ $\quad$ *WindowCursor*

3. $\quad 1 \leq UDCurY - OffsetY \leq WinHeight$ $\quad$ 2., *Doc9*

4. $\quad WindowLines\ WinCurY =$
$(WinMaskLines \text{ after } OffsetY)(UDCurY - OffsetY)$ $\quad$ 1., 3., propt. for

5. $\quad WindowLines\ WinCurY = WinMaskLines\ UDCurY$ $\quad$ 4., *Doc9*

6. $\quad (WindowLines\ WinCurY)\ WinCurX =$
$(((UDLines\ UDCurY) \text{ after } OffsetX) \text{ for } WinWidth)(UDCurX - OffsetX)$ $\quad$ 5., *Doc9*

7. $\quad 1 \leq WinCurX \leq WinWidth$ $\quad$ *WindowCursor*

8. $\quad 1 \leq UDCurX - OffsetX \leq WinHeight$ $\quad$ 7., *Doc9*

9. $\quad (WindowLines\ WinCurY)\ WinCurX =$
$((UDLines\ UDCurY) \text{after } OffsetX)(UDCurX - OffsetX)$ $\quad$ 6., 8., propt. for

10. $\quad (WindowLines\ WinCurY)\ WinCurX = (UDLines\ UDCurY)\ UDCurX$ $\quad$ 8., propt. after

■

We specify the initialization operation as:

$$Initialize_{Doc9} \;\widehat{=}\; Initialize_{Doc8} \;\wedge\; \Delta Doc9$$

which implies that initially both offsets are zero, and both window cursors are set to unity:

Lemma 3 : 8.2b

$Initialize_{Doc9}$

$\vdash$

$OffsetX', OffsetY' = 0, 0$

$WinCurX', WinCurY' = 1, 1$

■

Proof

1. $\quad UDCurX', UDCurY' = 1, 1$ $\quad$ Lemma 3 : 2.2d
2. $\quad WinCurX' : \mathbf{N}_1 \wedge WinCurY' : \mathbf{N}_1$ $\quad$ *Doc9'*
3. $\quad 1 - OffsetX' \geq 1 \wedge 1 - OffsetY' \geq 1$ $\quad$ 1., 2., *Doc9'*
4. $\quad OffsetX' : \mathbf{N} \wedge OffsetY' : \mathbf{N}$ $\quad$ *Doc9'*
5. $\quad OffsetX', OffsetY' = 0, 0$ $\quad$ 3., 4.
6. $\quad WinCurX', WinCurY' = 1, 1$ $\quad$ 1., 5.

■

Lemma 3 : 8.2c

$$Initialize_{Doc9}$$
$$\vdash$$
$$WindowLines' = \langle \langle \rangle \rangle$$

■

Proof

1. $UDLines' = \langle \langle \rangle \rangle$ — Lemma 3 : 2.2d
2. $\# WinMaskLines' = 1$ — 1., $Doc9'$
3. $WinMaskLines' = \langle (\langle \langle \rangle \rangle \text{ after } 0) \text{ for } WinWidth \rangle$ — 1., 2., Lemma 3 : 2.2b, $Doc9'$
4. $WinMaskLines' = \langle \langle \rangle \rangle$ — 3., $WinWidth : \mathbb{N}_1$
5. $WindowLines' = (\langle \langle \rangle \rangle \text{ after } 0) \text{ for } WinHeight$ — 4., Lemma 3 : 2.2b, $Doc9'$
6. $WindowLines' = \langle \langle \rangle \rangle$ — 5., $WinWidth : \mathbb{N}_1$

■

Thus we discharge PO 0.

8.2.1 An Operation To Centre The Window

We specify an operation to move the window vertically such that the current line appears in the centre of the window, document length permitting. We introduce:

$$HalfWinHeight : \mathbb{N} \mid HalfWinHeight = WinHeight / 2$$

(where "/" represents integer division, and so *HalfWinHeight* has minimum value *1* and maximum value *WinHeight*).

We require that *OffsetY* should be changed such that *WinCurY* equals *HalfWinHeight* - i.e. we set *OffsetY* to equal $(UDCurY - HalfWinHeight)$. In order to preserve the *Doc9* invariant that the offset be non-negative, we thus have the pre-condition:

$$UDCurY \geq HalfWinHeight$$

and so we define:

CenWin

$\Delta Doc9$
$\Xi Doc8$

$UDCurY \geq HalfWinHeight$
$OffsetX' = OffsetX$
$OffsetY' = UDCurY - HalfWinHeight$

---

together with the error message:

ErrorTooNearTop

$\Xi Doc9$
$rep! : Report$

$UDCurY < HalfWinHeight$
$rep! =$ "Too near top of document"

---

to give:

$$CentreWindow_{Doc9} \mathrel{\widehat{=}} (CenWin \land Success) \lor ErrorTooNearTop$$

The second disjunction does not change *Doc9*, and the first changes only *OffsetY*, the pre-condition ensuring that the invariant is preserved, and noting our comments in Section 8.2 that *Doc8* together with the offsets uniquely define *Doc9*, we discharge PO 1.

8.2.2 Promotion Of Doc8 Operations To The Doc9 State

Some of the *Doc8* operations will result in the cursor being moved to a position outside the current window, and the *Doc9* invariant requires that for such operations an appropriate window change is made in order to reposition the window to regain the cursor.

In general, if the operation leaves the cursor in the current window, it is desirable that there should be no window change, since a redisplay of the window in such cases would be both unnecessary, and tiresome for the user. However for some such operations the user would expect a window change (for example, *CursorDownPage*). therefore our promotion policy for a *Doc8* operation leaving the cursor in the window is non-deterministic, allowing a window change to be made.

We define an operation with pre-condition that the cursor is currently in the window, in which all *Doc8* components do not change but which allows the window offsets to change, providing that the new window position contains the cursor:

CursorInWindow

$\Delta WindowOffset$
$\Xi Doc8$

$UDCurX - OffsetX \in 1 \,..\, WinWidth$
$UDCurY - OffsetY \in 1 \,..\, WinHeight$
$UDCurX' - OffsetX' \in 1 \,..\, WinWidth$
$UDCurY' - OffsetY' \in 1 \,..\, WinHeight$

When an operation moves the cursor outside the current window, we change the offsets (and, necessarily, the window cursor, but leaving all other components of *Doc9* unchanged), but, clearly, for a given unbounded display there is more than one window change which will reposition the window to regain a "lost" cursor.

Although we are not concerned with the implementation of the window-policy for *Doc8* operations that leave the cursor outside the window, we stipulate that if the cursor can be regained by a *Scroll* (a change in the vertical offset only) or a *Pan* (a change in the horizontal offset only), then that should be the window reposition operation utilised (thus preserving the same screen columns or lines respectively, enabling the user to locate the screen cursor more easily). We define:

Scroll

$\Delta WindowOffset$
$\Xi Doc8$

$UDCurX - OffsetX \in 1 \,..\, WinWidth$
$UDCurY - OffsetY \notin 1 \,..\, WinHeight$
$OffsetX' = OffsetX$
$UDCurY' - OffsetY' \in 1 \,..\, WinHeight$

Pan

$\Delta WindowOffset$
$\Xi Doc8$

$UDCurX - OffsetX \notin 1 \,..\, WinWidth$
$UDCurY - OffsetY \in 1 \,..\, WinHeight$
$UDCurX' - OffsetX' \in 1 \,..\, WinWidth$
$OffsetY' = OffsetY$

It may not be possible to regain the cursor by either of these operations and so we define:

ScrollAndPan

$$\Delta WindowOffset$$
$$\Xi Doc8$$

$$UDCurX - OffsetX \notin 1 \,..\, WinWidth$$
$$UDCurY - OffsetY \notin 1 \,..\, WinHeight$$
$$UDCurX' - OffsetX' \in 1 \,..\, WinWidth$$
$$UDCurY' - OffsetY' \in 1 \,..\, WinHeight$$

---

to give a promotion operation which is the disjunction of these four window-change operations:

$$WindowPolicy \mathrel{\widehat=} \\ \quad (CursorInWindow \lor Scroll \lor Pan \lor ScrollAndPan) \land Doc9'$$

We note that the pre-conditions of the four disjunctions form a partition of the set of possible window states, and so the promotion operation is total: it satisfies the *Doc9* requirement that the cursor be in the window since each disjunction does; further, our comments of Section 8.2 indicate that *WindowLines* and the window cursor can be calculated from the operation. Thus *WindowPolicy* represents an operation in which none of the *Doc8* components may change, but allows the offsets (and hence *WindowLines* and the window cursor) to change in line with *Doc9*, and so post-sequential composition with the operation yields a state satisfying the *Doc9* invariant.

We use the set *EditOps4* of Section 7.6 to define the set of names:

$$EditOps5 \mathrel{\widehat=} EditOps4 \cup \{DownSearch, UpSearch, Replace\}$$

to give, for each operation *OP* in the set *EditOps5*:

$$\forall\ OP : EditOps5 \bullet OP_{Doc9} \mathrel{\widehat=} OP_{Doc8} \,;\, WindowPolicy$$

We thus discharge PO 1 for each *Doc8* operation.

PART FOUR

Hierarchical Refinement Of The Editor Specification

Contents

0 Introduction

In the abstract specification we make considerable use of redundancy. For example, we have equivalent "views" of an abstract state, and to specify an operation defined on that "Δ" state we select the view which is most appropriate (the correct update of the other views being ensured by the state invariant). In Section 1 we discuss abstract redundancy with respect to data refinement; we and also consider the implications of concrete redundancy in that section.

The specification is constructed in a hierarchical manner, each level of the hierarchy conforming to an abstract data type (comprising a state, an initialisation and a family of operations). This structure provides well-defined points which, in a natural way, break the specification into a number of smaller, more manageable parts, enabling the identification and discharge of proof obligations in a manner that follows the "separation of concerns" approach to software construction [1].

The goals of an abstract specification are not usually compatible with the design requirements of its implementation: the former seeks to express the relationship between the before- and after-states of a system rather than defining the algorithms underlying those relationships which the latter requires (the refinement calculus providing the bridge between the two). We can, however, use the problem-isolating structure of the specification to advantage in the refinement process.

We propose a novel hierarchical approach to refinement, in which we regard each specification hierarchy as a candidate for refinement. The implementation structure will thus be closely related to that of the specification.

We choose six abstract hierarchies on which to conduct refinement: the lowest-level hierarchy is the *Doc1* state (Section 2), followed by the *Doc3* state (Section 3), the *Doc4* state (Section 4), the *Doc6* state (Section 5), the *Doc8* state (Section 6), with the top-level hierarchy being the *Doc9* state (Section 7). We give reasons for choosing these particular hierarchies at the beginning of each section.

Each hierarchical refinement is based the refinement calculus that we present in Part 2. We first give the design decision, expressing the concrete-abstract relation through the *Rel* schema. Where necessary, we establish a theory relating to the design decision, enabling the subsequent refinement to proceed more smoothly. We discharge the data refinement proof obligation by calculation of *AbsRel*, and consider concrete state reconfiguration by calculating *ConcRel*. Variables introduced in *Rel* form the global variables of the implementation.

We then turn to operational refinement, our starting point for which is the calculation of the weakest concrete operation, possibly on a specific configuration of the concrete state. In the majority of cases we then apply the rules developed in the calculus in a stepwise manner to achieve the refinement (rather than writing down what we feel is the refinement and proving that it is so from our definition).

On each level selected for refinement we refine sufficient operations to indicate the method of refinement for all operations specified on that hierarchy. We then give the promotion method for operations that have already been refined on lower-level hierarchies, (and since each abstract operation is usually promoted in the same way, we need only provide one promotion operation).

For convenience we give a summary of the concrete state hierarchies in Appendix B.

0.1 A Note On Refinement Convention

We use the following steps in the data refinement of each abstract data type (hierarchy):

The design decision

A statement of the concrete representation of the abstract state, together with an explanation of why the representation was chosen.

The concrete-abstract invariant

we give the relationship between the abstract and concrete states, *Rel*, together with the calculation of the schemas *AbsRel* and *ConcRel*; we use the former to prove that the design decision is adequate (or that we have an ∞-implementation), and the latter to define the equivalence class of concrete states corresponding to a single abstract state. We may refine this latter schema to a concrete reorganising operation $Rel_{specific}$.

The following steps are used in the refinement of each operation (several may be combined into a single step):

Specification

For convenience we give the abstract specification, given in Part 3.

Expansion

The expansion of the abstract specification (usually into vertical schema form).

Weakest concrete operation

Using the results established in Part 2, Section 2, we replace the abstract state components by their concrete counterparts (through *Rel*).

Weakest Standard concrete operation

As above, but specific concrete counterparts (through $Rel_{specific}$) are used to obtain a particular concrete configuration.

Simplification

The weakest concrete operation is obtained by textual replacement, and can usually be considerably simplified before refinement proceeds. We establish a theory relating to the design decision to aid this process of simplification.

Refinement

We use the results established in Part 2, Section 3, indicating which we use by, for example, [$\sqsubseteq$ 2 : 3.1*a*].

Code

We use the results established in Part 2, Section 3.5. When refining to a loop we incorporate the invariant predicate, variant function and guard negation as **{** *assertions* **}** to aid the discharge of the proof obligation, and use a subscripted "*o*" to indicate initial values.

It will not be necessary to include each of the above steps in the refinement of every operation; many operations follow a similar development pattern and we avoid repetition where possible by combining several steps into one. Where necessary we supplement the refinement process with notes on one or more of the above steps (appearing after the last refinement step, so as not to clutter the development).

Use of shorthand notation

Because of the hierarchical nature of the refinement process, often components not relating to the current hierarchy will not change (since, for example, many operations are promoted by the maintenance of a no-change state), and as the hierarchical level increases, the number of unchanging lower-level components may be considerable. When we are using a schema notation, we may, of course, employ the "Ξ" no-change schema. During the latter stages of each refinement we wish to avoid needless repetition of the signature declaration, and we give only the predicate part of the schema, enclosed by a square bracket "[", and to avoid the "formal clutter" [13] that would ensue from a long list of unchanged components, we use the convention that, for example, "*NoChange*(*ConcDocN* \ *comp1*, *comp2*)" implies that each component of *ConcDocN* except *comp1* and *comp2* will not change during that refinement step.

We continue to use the convention of vertical alignment of predicates to imply logical conjunction (Part 2, Section 0.1).

Further, we use :

$a{+}{+}$ and $a{-}{-}$

to mean, respectively :

$a := (a+1)$ and $a := (a-1)$

and we again use, for example, "*string*" to represent $< s, t, r, i, n, g >$.

Input and output conventions

If, for example, operation A takes input parameters $x_1?, x_2? \ldots x_n?$, we use:

$A(value_1, value_2 \ldots value_n)$

to indicate that each $x_i?$ is set to $value_i$ by the operation with which A is pre-sequentially composed, when there is no possibilty of confusion.

Further, if, for example, an operation sets the value of its (single) output parameter *y!* to *value*, we use *return(value)* (typically, this will be used for the return of the report indicating the success or otherwise of the operation).

0.2 Overview Of The Command Loop Structure

The command loop can be regarded as the highest level hierarchy of the implementation, immediately above *Doc9*, and it acts as a filter mechanism to identify keys which are not bound to specific editor operations. When a command is entered at the keyboard, the keyboard interpret routine ("kbdinterpret.c", Appendix C) will set the global variable *OP* to the name of one of the editor operations, or *NotImplemented* should the key not be bound to an operation (see "consts.c", Appendix C); if the operation is *InsertChar*, the global *OPChar* will be set to the character inserted.

If *OP* is *NotImplemented*, an appropriate error message will be displayed and control stays within the loop structure, otherwise control is passed to *ConcDoc9*. If the operation is specified on that level, it is effected, the display (if necessary) updated, and control passed back to the command loop.

If the operation is not specified on that hierarchy, control passes down through the hierarchies until it reaches that on which it is specified (the filtering during the command loop ensuring that a hierarchy will be found). The operation is effected and the report (*rep* - see "consts.c", Appendix C) passed back up through the hierarchical structure (with the "promotion" mechanisms being applied, details of which may be found in the relevant refinements, Sections 2 to 7). So, for example, if the n^{th} hierarchy receives a *rep* passed from OP_{n-1} and has promotion mechanism $Promote_n$, the code is:

$$(rep \ := \ OP_{n-1}) \ ; \ Promote_n \ ; \ return(rep)$$

When the report eventually reaches *ConcDoc9*, the display is (if necessary) updated, and if the operation was not successful an appropriate message displayed before control is passed back from *ConcDoc9* to the command loop. In general, unsuccessful operations will require no amendment to the display but will necessitate the reporting of an appropriate error message (see "prompt.c", Appendix C).

A further global variable, *OPType*, is set (in the implementation of *Doc1*) for each operation, the classification being *LeftMove*, *RightMove*, *NoMove*, *LeftDelete*, *RightDelete*, *LeftInsert* or *RightInsert*. We do this since a group of operations (for example the three left delete operations of *Doc1*) are treated in exactly the same way by, for example, the re-display algorithm of *ConcDoc9*, and it is more convenient to use the operation type rather than the operation itself. (This would also keep algorithmic changes down to a minimum should further operations be added to the specification at a later date.)

1 Specification And Concrete Redundancy Considerations

The abstract state *Doc1* comprises the components $Pair_{Char}$, $Pair_{Word}$ and $Pair_{Line}$ together with an invariant relationship which renders any two of the three components redundant, in the sense that they may be computed from the third.

We are at liberty to exclude such redundant components from the design, since we may obtain complete representation without them. However, efficiency considerations may dictate that such items are best included in the implementation (thereby, for example, obviating the need for the continual re-calculation of a particular value). It is worth noting that a redundant component included in the design at a particular stage in the development, and subsequently found not to be required, may be removed from the implementation by methods of program transformation [26].

Of course the inclusion of redundant components will not violate our concept of refinement, since its *Safety* aspect [$\sqsubseteq$ 2 : 3.2*a*] explicitly allows us to "do more" (i.e. to be more deterministic) than the specification requires, provided we introduce no conflict with those requirements, and, therefore, we are at liberty to include further concrete components having no abstract counterpart.

2 Refinement Of Doc1

The *Doc1* hierarchy includes sixteen operations which we may regard as relating to memory management, and we choose it as our lowest-level hierarchy on which to conduct refinement.

2.1 The Design Decision

We need to represent only one *Doc1* abstract view, and we choose to represent the $Pair_{Char}$ component by a character array, *Arr* (assumed to have a maximum length of *Max*, a natural number *ResourceLimit* dependent upon available memory size):

$$Max : ResourceLimit$$

$$CharArray \mathrel{\widehat=} [\ Arr : 1 \,..\, Max \;\longrightarrow\; Char]$$

together with the pointers *LP* (*L*eft *P*ointer), *RP* (*R*ight *P*ointer), and *CP* (*C*ursor *P*ointer). The contents of the array from *1* to *LP* and from $(RP + 1)$ to *Max* represent the concatenation of the left and right character sequences, with *CP* providing the current position. Thus we require that *LP* may not exceed *RP* and that *CP* must not exceed the length of the array contents:

$$
\begin{array}{|l}
\hline
\textit{Pointers} \\
LP, RP, CP : 0 \ldots Max \\
\hline
LP \leq RP \\
CP \leq Max + LP - RP \\
\hline
\end{array}
$$

to give the concrete state:

$$ConcDoc1 \mathrel{\widehat{=}} CharArray \wedge Pointers$$

This general configuration will be used for the refinement of cursor-movement operations, which may therefore be accomplished by a change of CP (since the contents of the document and, hence, of the array will not change), thus avoiding unnecessary "array shuffling".

However, we also consider the particular configuration of the array in which the contents from 1 to LP correspond to the left character sequence (and thus CP will be equal to LP), and the array contents from $(RP + 1)$ to Max correspond to the right character seqnence. Changes made to the document will take place at the current position, and so this configuration will be used for the refinement of operations that change the content of the document; for example the left insertion of characters will commence at array position $(LP + 1)$, with LP and CP being incremented accordingly.

We thus define the *Standard* concrete state:

$$ConcDoc1_{Standard} \mathrel{\widehat{=}} [\ ConcDoc1 \mid CP = LP\ ]$$

2.1.1 The Concrete-Abstract Invariant

The content of the document is represented by the part of the array from 1 to LP and $(RP + 1)$ to Max, with CP equal to the length of the left character sequence. We therefore have:

$$Arr \text{ for } LP \frown Arr \text{ after } RP = Left_{Char} \frown Right_{Char}$$
$$\# Left_{Char} = CP$$

For ease of reference, we define:

$$ArrCont \mathrel{\widehat{=}} Arr \text{ for } LP \frown Arr \text{ after } RP$$

and then an equivalent specificatiou, in which the left and right sequences are explicitly defined, is:

$$
\begin{array}{lcl}
Left_{Char} & = & ArrCont \text{ for } CP \\
Right_{Char} & = & ArrCont \text{ after } CP
\end{array}
$$

to give:

$$\begin{array}{|l}
\multicolumn{1}{l}{Rel_{Doc1}} \\
\hline
Doc1 \\
ConcDoc1 \\
\hline
Left_{Char} = ArrCont \text{ for } CP \\
Right_{Char} = ArrCont \text{ after } CP \\
\hline
\end{array}$$

We show that Rel relates each concrete state to a valid abstract state; we have, by definition,

$$\begin{array}{lcl}
Left_{Char} & \widehat{=} & (Arr \text{ for } LP \frown Arr \text{ after } RP) \text{ for } CP \\
Right_{Char} & \widehat{=} & (Arr \text{ for } LP \frown Arr \text{ after } RP) \text{ after } CP
\end{array}$$

and the definitions of Arr, **for**, **after** and "$\frown$" ensure that both are valid character sequences. Clearly, for a given concrete state both character sequences will be unique, and so we establish:

Lemma 4 : 2.1.1a

$$Rel_{Doc1}$$
$$\vdash$$
$$\forall\ ConcDoc1 \bullet \exists_1\ Doc1 \bullet Rel_{Doc1}$$

■ .

We now calculate $AbsRel$ and have, after simplification:

$$AbsRel_{Doc1} \;\widehat{=}\; [\ \Xi Doc1 \mid \#(Left_{Char} \frown Right_{Char}) \leq Max\ ]$$

and since:

$$\lim_{Max \to \infty} (\#(Left_{Char} \frown Right_{Char}) \leq Max) \;\equiv\; \text{true}$$

we discharge our data refinement proof obligation by appealing to [$\sqsubseteq$ 2 : 4.1b], because Lemma 4:2.1.1a together with this result imply an ∞-refinement:

Lemma 4 : 2.1.1b

$$\vdash$$
$$Doc1 \sqsubseteq_\infty ConcDoc1$$

■

We now establish some theory relating to the design. Firstly, if the array content is empty, LP and CP are both zero and RP is equal to Max:

Lemma 4 : 2.1.1c

$$
\begin{array}{l}
\quad ConcDoc1 \;|\; ArrCont = \langle\rangle \\
\vdash \\
\quad LP = 0 \;\wedge\; RP = Max \;\wedge\; CP = 0
\end{array}
$$

■

Proof

1.	Arr for $LP = Arr$ after $RP = \langle\rangle$	defn. $ArrCont$, propt. " $\frown$ "
2.	$LP = 0$	1., propt. "for"
3.	$RP \geq Max$	1., propt. "after"
4.	$RP = Max$	3., $RP : 0 \mathinner{..} Max$
5.	$CP \leq 0$	2., 4., $CP \leq Max + LP - RP$
6.	$CP = 0$	2., 5., $CP : 0 \mathinner{..} Max$

■

Secondly, we note that LP and RP are pointers to Arr, whereas CP is a pointer to $ArrCont$, and we give the following lemmas (the proofs of which follow immediately from the definitions of "for", "after" and "∪") which relate $ArrCont$ to Arr:

Lemma 4 : 2.1.1d

$$
\begin{array}{llll}
\multicolumn{4}{l}{\quad ConcDoc1} \\
\multicolumn{4}{l}{\vdash} \\
\multicolumn{4}{l}{\;\forall\, ptr : 0 \mathinner{..} Max \bullet} \\
\quad ptr \leq LP & \Leftrightarrow & ArrCont\ ptr = Arr\ ptr \\
\quad ptr \leq LP & \Leftrightarrow & ArrCont \text{ for } ptr = Arr \text{ for } ptr \\
\quad ptr \leq LP & \Leftrightarrow & ArrCont \text{ after } ptr = (ptr + 1 \mathinner{..} LP \;\cup\; RP + 1 \mathinner{..} Max) \upharpoonleft Arr \\
\quad ptr > LP & \Leftrightarrow & ArrCont\ ptr = Arr\ (ptr + RP - LP) \\
\quad ptr > LP & \Leftrightarrow & ArrCont \text{ for } ptr = (1 \mathinner{..} LP \;\cup\; RP + 1 \mathinner{..} ptr) \upharpoonleft Arr \\
\quad ptr > LP & \Leftrightarrow & ArrCont \text{ after } ptr = Arr \text{ after } ptr
\end{array}
$$

■

Lemma 4 : 2.1.1e

$$\begin{array}{l} ConcDoc1 \\ \vdash \\ \forall\ ptr1, ptr2 : 0 \mathrel{..} Max \bullet \\ \quad ptr2 \leq LP \Leftrightarrow \\ \qquad ArrCont \llparenthesis ptr1 \mathrel{..} ptr2 \rrparenthesis = Arr \llparenthesis ptr1 \mathrel{..} ptr2 \rrparenthesis \\ \quad ptr1 \leq LP < ptr2 \Leftrightarrow \\ \qquad ArrCont \llparenthesis ptr1 \mathrel{..} ptr2 \rrparenthesis = Arr \llparenthesis ptr1 \mathrel{..} LP \cup RP + 1 \mathrel{..} ptr2 + RP - LP \rrparenthesis \\ \quad ptr1 > LP \Leftrightarrow \\ \qquad ArrCont \llparenthesis ptr1 \mathrel{..} ptr2 \rrparenthesis = Arr \llparenthesis ptr1 + RP - LP \mathrel{..} ptr2 + RP - LP \rrparenthesis \end{array}$$

■

2.2 Initialization

Abstract specification:

$$\begin{array}{|l} \textit{Initialize}_{Doc1} \\ \hline \Delta Doc1 \\ \hline Left_{Char}{}' = Right_{Char}{}' = <> \end{array}$$

Weakest concrete operation:

$$\begin{array}{|l} \textit{Initialize}_{Doc1}\ \mathbf{C} \\ \hline \Delta ConcDoc1 \\ \hline ArrCont' \ \mathsf{for}\ CP' = ArrCont' \ \mathsf{after}\ CP' = <> \end{array}$$

Simplification:

$$\left[\ LP', CP', RP' = 0, 0, Max \right.$$

Code:

$Initialize_{Doc1}$

$$LP := 0 \; ; \; CP := 0 \; ; \; RP := Max$$

Notes
Simplification:
The definition of for implies that either $ArrCont'$ is empty or CP' is 0, and that for

after implies that either $ArrCont'$ is empty or CP' has value greater than the length of $ArrCont$, which together imply that $ArrCont'$ is empty. The simplification then follows from [$\sqsubseteq$ 4 : 2.1.1c].

2.3 Content-Changing Operations And The Standard State

As discussed in Section 2.1, operations which change the document's content will be refined on $ConcDoc1_{Standard}$. We have:

$$Rel_{Doc1\,Standard} \mathrel{\widehat{=}} [\ Rel_{Doc1} \mid CP = LP\ ]$$

which expands to:

$Rel_{Doc1\,Standard}$

$Doc1$
$ConcDoc1$

$Left_{Char} = Arr$ **for** LP
$Right_{Char} = Arr$ **after** RP

in which $Left_{Char}$ and $Right_{Char}$ are directly related to Arr (rather than indirectly as in $ArrCont$).

We now calculate $ConcRel$, and have, after simplification:

$ConcRel_{Doc1}$

$\Delta ConcDoc1$

$CP' = CP$
$ArrCont' = ArrCont$

and may informally interpret this equivalence class as those concrete states whose cursor pointers are the same, and whose arrays agree once the array positions not being used have been filtered out. Thus LP and RP will not be uniquely defined, although their numerical difference must be the same for each equivalence class (equal to the difference between Max and the length of $ArrCont$).

Using the $Standard$ configuration, we calculate $ConcRel_{Doc1\,Standard}$ to get:

$$\begin{array}{|l}
\hline
\textit{ConcRel}_{Doc1\,Standard} \\
\hline
\Delta ConcDoc1 \\
\hline
CP' = CP \\
LP' = CP \\
ArrCont' = ArrCont \\
\hline
\end{array}$$

By [$\sqsubseteq 2:2.2b$] we may choose our starting point for refining a content-changing abstract operation AOP_{Doc1} as:

$$ConcRel_{Doc1\,Standard} \;;\; AOP_{Doc1}\mathbf{C}_{Standard}$$

where $AOP_{Doc1}\mathbf{C}_{Standard}$ is obtained from the abstract operation by the substitution of the concrete state conforming to $Rel_{Doc1\,Standard}$ for the before-state, and that conforming to Rel_{Doc1} for the after-state variables.

In fact we choose to use the *Standard* concrete configuration for the after-variable substitution as well, which we denote by $AOP_{Doc1}\mathbf{C}_{Standard'}$ noting that in so doing we are not changing the pre-condition, and that the *Standard* configuration implies the non-*Standard*, and so we can appeal to [$\sqsubseteq 2:3.4d$] to obtain:

$$ConcRel_{Doc1\,Standard} \;;\; AOP_{Doc1}\mathbf{C}_{Standard} \;\sqsubseteq\; ConcRel_{Doc1\,Standard} \;;\; AOP_{Doc1}\mathbf{C}_{Standard'}$$

2.3.1 The "Standardize" Reconfigure Operation

We partition the concrete states into those where CP exceeds LP, those where LP exceeds CP and those where they are the same. For the first of these we must move the "gap" (the portion of the array that is not used) to the left to achieve a *Standard* state:

$$\begin{array}{|l}
\hline
\textit{MoveGapLeft} \\
\hline
\Delta ConcDoc1 \\
\hline
LP > CP \\
LP' = CP' = CP \\
ArrCont' = ArrCont \\
\hline
\end{array}$$

For the second the gap must be moved to the right:

$$\begin{array}{|l}
\hline
\textit{MoveGapRight} \\
\hline
\Delta ConcDoc1 \\
\hline
LP < CP \\
LP' = CP' = CP \\
ArrCont' = ArrCont \\
\hline
\end{array}$$

and for the third case there is nothing to do:

$$Standardized \mathrel{\widehat{=}} [\ \Xi ConcDoc1 \mid CP = LP\ ]$$

We now define:

$$Standardize_{Doc1} \mathrel{\widehat{=}} MoveGapLeft \vee MoveGapRight \vee Standardized$$

To show that $Standardize_{Doc1}$ refines $ConcRel_{Doc1\,Standard}$ we appeal to [$\sqsubseteq$ 2 : 3.3a], noting that the disjunct of the pre-conditions of the former is true (and by pre.1 so is the pre-condition of the disjunct, and thus *Domain* is satisfied), and that each disjunct contains the post-condition that *ArrCont* and *CP* do not change (and thus *Safety* is satisfied).

By now appealing to [$\sqsubseteq$ 2 : 3.2c], [$\sqsubseteq$ 2 : 3.4c] and [$\sqsubseteq$ 2 : 2.2b] we establish:

Lemma [$\sqsubseteq$ 4 : 2.3.1a]

$\vdash$

$$AOP_{Doc1} \sqsubseteq Standardize_{Doc1}\ ;\ AOP_{Doc1}\mathbf{C}_{Standard'}$$

■

2.3.2 Refinement Of "Standardize"

Specification :

$$Standardize_{Doc1} \mathrel{\widehat{=}} MoveGapLeft \vee MoveGapRight \vee Standardized$$

Expansion:

$$\left[\begin{array}{l}
LP' = CP' = CP \\
RP' = RP - LP + CP \\
Arr' \text{ for } CP = Arr \text{ for } CP \\
Arr' \text{ after } RP = Arr \text{ after } RP \\
\quad LP > CP \\
\quad \text{pred}^{\,RP-LP}\ ;\ CP + 1 \mathinner{..} LP \lhd Arr = RP' + 1 \mathinner{..} RP \lhd Arr' \\
\vee \\
\quad LP < CP \\
\quad \text{succ}^{\,RP-LP}\ ;\ RP + 1 \mathinner{..} RP' \lhd Arr = LP + 1 \mathinner{..} CP \lhd Arr' \\
\vee \\
\quad LP = CP \\
\quad CP + 1 \mathinner{..} LP \lhd Arr = CP + 1 \mathinner{..} LP \lhd Arr'
\end{array}\right.$$

Refinement:

$$
\left[\begin{array}{l}
LP' = CP' = CP \\
RP' = RP - LP + CP \\
\quad LP > CP \\
\quad Arr' = Arr \oplus \textsf{pred}^{\,RP-LP} \,;\, CP + 1 \,..\, LP \lhd Arr \\
\vee \\
\quad LP < CP \\
\quad Arr' = Arr \oplus \textsf{succ}^{\,RP-LP} \,;\, RP + 1 \,..\, RP' \lhd Arr \\
\vee \\
\quad LP = CP \\
\quad Arr' = Arr
\end{array}\right.
$$

Code for $Standardize_{Doc1}$

```
if
 | (LP > CP) -->
 |     do
 |      | (LP > CP) --> Arr RP := Arr LP ; RP-- ; LP--
 |      |  { Invariant : ArrCont = ArrCont_o }
 |      |  { Invariant : Arr = Arr_o ⊕ pred ^(RP-LP) ; LP + 1 .. LP_o ◁ Arr_o }
 |      |  { Variant : LP - CP }
 |      |  { Guard Negation : LP ≤ CP }
 |     od
[]
 | (LP < CP) -->
 |     do
 |      | (LP < CP) --> Arr(LP + 1) := Arr(RP + 1) ; RP++ ; LP++
 |      |  { Invariant : ArrCont = ArrCont_o }
 |      |  { Invariant : Arr = Arr_o ⊕ succ ^(RP-LP) ; RP_o + 1 .. RP ◁ Arr_o }
 |      |  { Variant : CP - LP }
 |      |  { Guard Negation : LP ≥ CP }
 |     od
fi
```

Notes

We consider *MoveGapLeft*; similar comments apply to *MoveGapRight*.

Expansion and Refinement:

We pursue refinement on each disjunct, appealing to [$\sqsubseteq$ 2 : 3.3*b*], and noting that we do not change each pre-condition we satisfy *Domain*. Each array element moved will be moved a distance equal to the array gap (i.e. $RP - LP$), and the predicate:

$$Arr' = Arr \oplus \textsf{pred}^{\,RP-LP} \,;\, (CP + 1) \,..\, LP \lhd Arr$$

implies the second predicate of the disjunct in the expansion, since

$$\textsf{pred}^{RP-LP} \; ; (CP + 1)\,..\,LP \quad = $$
$$(CP + 1 + RP - LP)\,..\,(LP + RP - LP) \quad = $$
$$RP' + 1\,..\,RP$$

so

$$CP + 1\,..\,LP \lhd Arr \;=\; RP' + 1\,..\,RP \lhd Arr'$$

and since Arr' does not change up to CP and after RP, *ArrCont* will remain unchanged, and hence we satisfy *Safety* of [$\sqsubseteq$ 2 : 3.3*b*].

Code:

The pre-condition for the loop body is that both LP and RP must be non-zero: the former is implied by the guard, since CP cannot be less than zero, and the latter by the requirement that RP must be at least as large as LP; thus the guard implies the pre-condition for the loop body.

The invariant is initially true since $LP + 1\,..\,LP$ is empty, and the guard negation implies the post-condition of the specification expansion. Each iteration of the body decrements both pointers, which means that $RP' + 1\,..\,RP$ is extended by one at its left hand end, and it copies the contents of location LP into location RP before they are decreased, and thus invariant is re-established.

Finally we again appeal to [$\sqsubseteq$ 2 : 3.5.2*b*] to refine to the alternative command, and [$\sqsubseteq$ 2 : 3.5.3*a*] to ensure that the disjunct of the refinement can be safely replaced with the loop.

2.3.3 Refinement Of "LeftDeleteChar"

Specification :

$$(LeftDelChar \wedge Success) \vee ErrorTopOfDoc$$

Expansion:

LeftDeleteChar$_{Doc1}$

$\Delta Doc1$

$Right_{Char}' = Right_{Char}$

$\quad Left_{Char} \neq \langle\rangle$

$\quad Left_{Char}' = \textsf{front}\ Left_{Char}$

$\quad rep! = $ "OK"

$\vee$

$\quad Left_{Char}' = Left_{Char} = \langle\rangle$

$\quad rep! = $ "At top of document"

Weakest *Standard'* concrete operation:

$$\begin{array}{l}
LeftDeleteChar_{Doc1}C_{Standard'} \\
\hline
\Delta ConcDoc1_{Standard} \\
\hline
Arr' \text{ after } RP' = Arr \text{ after } RP \\
\quad Arr \text{ for } LP \neq <> \\
\quad Arr' \text{ for } LP' = \text{front } (Arr \text{ for } LP) \\
\quad rep! = \text{"OK"} \\
\vee \\
\quad Arr' \text{ for } LP' = Arr \text{ for } LP = <> \\
\quad rep! = \text{"At top of document"}
\end{array}$$

Refinement:

$$\left[\begin{array}{l}
LP, LP' = CP, CP' \\
Arr', RP', = Arr, RP \\
\quad LP \neq 0 \\
\quad LP', CP' = (LP - 1), (CP - 1) \\
\quad rep! = \text{"OK"} \\
\vee \\
\quad LP = LP' = CP' = CP = 0 \\
\quad rep! = \text{"At top of document"}
\end{array}\right.$$

Code for $LeftDeleteChar_{Doc1}$

$$\begin{array}{l}
Standardize_{Doc1} \; ; \\
\{ LP = CP \} \\
\textbf{if} \\
\mid (LP \neq 0) \longrightarrow LP-- \; ; CP-- \; ; return(\text{"OK"}) \\
\square \\
\mid (LP = 0) \longrightarrow return(\text{"At top of document"}) \\
\textbf{fi}
\end{array}$$

Notes

Refinement:

We use [$\sqsubseteq$ 2 : 3.3*b*] which allows us to treat the operation as a disjunct, noting that the pre-condition of the two disjuncts does not change *Domain*; although the operation stipulates that the array up to $(LP - 1)$ and after RP does not change, we choose to leave the entire array unchanged and appeal to [$\sqsubseteq$ 2 : 3.1*a*] - we are strengthening the post-condition - to establish *Safety* of [$\sqsubseteq$ 2 : 3.3*b*].

Code:

We use [4 : 2.3.1*a*] to give a *Standard* refinement, and appeal to [2 : 3.5.2*b*] to produce the **if ... fi** construct: we note that the guards should, in fact, test both LP and CP, but since we are sure that the two pointers are equal before the operation starts (because of the post-condition of *Standardize*), we choose to test for just one (and so in the first conjunct, the guard does imply the pre-condition of the body).

2.3.4 Refinement Of "InsertChar"

Specification :

$$((InsNonTab \lor InsTab) \land Success) \lor ErrorFull$$

Expansion:

$$
\begin{array}{l}
InsertChar_{Doc1} \\
\hline
\Delta Doc1 \\
OPChar? : Char \\
\hline
Right_{Char}' = Right_{Char} \\
\qquad\qquad OPChar? \neq tab \\
\qquad\qquad Left_{Char}' = Left_{Char} \frown \langle OPChar? \rangle \\
\qquad \lor \\
\qquad\qquad OPChar? = tab \\
\qquad\qquad Left_{Char} \text{ prefix } Left_{Char}' \\
\qquad\qquad \mathrm{rng}\ (Left_{Char}' - Left_{Char}) = \{sp\} \\
\qquad rep! = \text{"OK"} \\
\lor \\
\qquad Left_{Char}' = Left_{Char} \\
\qquad rep! = \text{"Editor full"}
\end{array}
$$

Weakest *Standard* concrete operation:

$InsertChar_{Doc1} \mathbf{C}_{Standard}$

$$
\begin{array}{l}
\Delta ConcDoc1_{Standard} \\
OPChar? : Char \\
\hline
Arr' \text{ after } RP' = Arr \text{ after } RP \\
\qquad\qquad OPChar? \neq tab \\
\qquad\qquad Arr' \text{ for } LP' = (Arr \text{ for } LP) \frown \langle OPChar? \rangle \\
\qquad \vee \\
\qquad\qquad OPChar? = tab \\
\qquad\qquad (Arr \text{ for } LP) \text{ prefix } (Arr' \text{ for } LP') \\
\qquad\qquad \text{rng } ((Arr' \text{ for } LP') - (Arr \text{ for } LP)) = \{sp\} \\
\qquad rep! = \text{“OK”} \\
\quad \vee \\
\qquad Arr' \text{ for } LP' = Arr \text{ for } LP \\
\qquad rep! = \text{“Editor full”}
\end{array}
$$

Refinement and simplification:

$$
\left[
\begin{array}{l}
LP, LP' = CP, CP' \\
RP' = RP \\
\qquad LP \neq RP \\
\qquad\qquad OPChar? \neq tab \\
\qquad\qquad LP' = LP + 1 \\
\qquad\qquad Arr' = Arr \oplus \{(LP + 1) \mapsto OPChar?\} \\
\qquad \vee \\
\qquad\qquad OPChar? = tab \\
\qquad\qquad LP + 1 \,..\, LP' \lhd Arr' = LP + 1 \,..\, LP' \lhd Arr \\
\qquad\qquad Arr \,(\!|\, LP + 1 \,..\, LP' \,|\!)\, = \{sp\} \\
\qquad rep! = \text{“OK”} \\
\vee \\
\qquad LP = RP = LP' \\
\qquad Arr' = Arr \\
\qquad rep! = \text{“Editor full”}
\end{array}
\right.
$$

Code for $InsertChar_{Doc1}(OPChar)$ $OPChar? : Char$

$ptr := CP : 0 \,..\, Max$;
$count := 0 : 0 \,..\, Max$;
$Standardize_{Doc1}$;
$\{ LP = CP \}$
if
| $(LP \neq RP \wedge OPChar \neq tab) \longrightarrow LP{+}{+} ; CP{+}{+} ;$
| $Arr\ LP := OPChar$; *return*("OK")
□
| $(LP \neq RP \wedge OPChar = tab) \longrightarrow$ *InsertSpaces* ; *return*("OK")
□
| $(LP = RP) \longrightarrow$ *return*("Editor full")
fi

Code for *InsertSpaces*

do
| $(ptr \neq 0 \wedge Arr\ ptr \neq nl) \longrightarrow count{+}{+} ; ptr{-}{-}$
| { Invariant : $nl \notin Arr \llparenthesis\, ptr + 1 \,..\, CP \,\rrparenthesis$ }
| { Variant : ptr }
| { Guard Negation : $ptr = 0 \vee Arr\ ptr = nl$ }
od ;
$count := tabstop - (count \% tabstop)$;
$\{ (LP_o + count) \% tabstop = 0 \}$
do
| $(count \neq 0 \wedge LP \neq RP) \longrightarrow Arr(LP + 1) := sp ;$
| $CP{+}{+} ; LP{+}{+} ; count{-}{-}$
| { Invariant : $LP \neq LP_o \Rightarrow Arr \llparenthesis\, LP_o + 1 \,..\, LP \,\rrparenthesis = \{sp\}$ }
| { Variant : $count$ }
| { Guard Negation : $count = 0 \vee LP = RP$ }
od

Notes

Refinement:

We again use [$\sqsubseteq$ 4 : 2.3.1*a*] to proceed on a $ConcDoc1_{Standard}$ state, and thus each disjunct includes the invariants:

$$\# ArrCont \leq Max$$
$$\# ArrCont' \leq Max$$

which, for a successful insert operation, imply:

$$\# ArrCont \quad < \quad Max$$

and this is the pre-condition for the first disjunct. In order to keep the operation total we introduce the pre-condition for the second disjunct:

$$\# ArrCont \; = \; Max$$

and appeal to [$\sqsubseteq$ 2 : 3.2*b*], the two pre-conditions simplifying to:

$$LP \neq RP$$
$$LP = RP$$

In the case of the *tab* character, the abstract specification, of course, does not stipulate how many space characters should be inserted. We take the design decision that after the operation, the number of characters between the cursor and the previous newline (or the start of the document, if no such character exists) is an exact multiple of *tabstop*; the exception to this is when the editor's capacity will not allow all such space characters to be inserted, in which case as many spaces as possible are inserted (and the "OK" report issued).

Code:

We are able to refine to the alternate construct by virtue of [$\sqsubseteq$ 2 : 3.5.2*b*], noting that if *LP* is not equal to *RP* then the *ConcDoc1* invariant that *LP* does not exceed *RP*, together with the signature of *RP*, implies that *LP* is less than *Max*. Further, since *Standardize* sets *LP* to *CP*, the latter must also be less than *Max*. Thus each guard ensures the pre-condition for its body. The loops follow from [$\sqsubseteq$ 2 : 3.5.3*a*], and we are able to decompose *InsertSpaces* to the sequential composition of two loops by appealing to [$\sqsubseteq$ 2 : 3.4*b*] noting that every loop represents a total operation (and so the two *Domain* conditions follow). *Safety* following from the second invariant and guard negation, noting that the loop will iterate at least once (by definition of "%" - the operator such that $(a\%b)$ gives the remainder when a is divided by b).

2.3.5 Refinement Of "RightDeleteWord"

Specification :

$$(RightDelWord \wedge Success) \;\vee\; ErrorBotOfDoc$$

Expansion:

$RightDeleteWord_{Doc1}$

$\Delta Doc1$

$Left_{Word}' = Left_{Word}$

$\quad Right_{Word} \neq \langle\rangle$

$\quad Right_{Word}' = \text{tail } Right_{Word}$

$\quad rep! = \text{"OK"}$

$\vee$

$\quad Right_{Word} = Right_{Word}' = \langle\rangle$

$\quad rep! = \text{"At bottom of document"}$

Weakest *Standard* concrete operation:

$RightDeleteWord_{Doc1}\mathbf{C}_{Standard}$

$\Delta ConcDoc1_{Standard}$

$\mathsf{FW}^{-1}(Arr' \text{ for } LP') = \mathsf{FW}^{-1}(Arr \text{ for } LP)$

$\quad \mathsf{FW}^{-1}(Arr \text{ after } RP) \neq \langle\rangle$

$\quad \mathsf{FW}^{-1}(Arr' \text{ after } RP') = \text{tail } (\mathsf{FW}^{-1}(Arr \text{ after } RP))$

$\quad rep! = \text{"OK"}$

$\vee$

$\quad \mathsf{FW}^{-1}(Arr \text{ after } RP) = \mathsf{FW}^{-1}(Arr' \text{ after } RP') = \langle\rangle$

$\quad rep! = \text{"At bottom of document"}$

Simplification and refinement:

$Arr = Arr'$

$LP' = LP = CP = CP'$

$\quad RP \neq Max$

$\quad \mathsf{FW}^{-1}(Arr' \text{ after } RP' = \text{tail } (\mathsf{FW}^{-1}(Arr \text{ after } RP))$

$\quad rep! = \text{"OK"}$

$\vee$

$\quad RP = RP' = Max$

$\quad rep! = \text{"At bottom of document"}$

Further simplification:

$$
\left[\begin{array}{l}
Arr = Arr' \\
LP' = LP = CP = CP' \\
\quad RP \neq Max \\
\quad\quad Arr(RP+1) = nl \\
\quad\quad RP' = RP + 1 \\
\quad \vee \\
\quad\quad Arr \,(\!| RP+1 \,..\, RP' |\!)\, \subseteq \{sp\} \\
\quad\quad RP' \neq Max \;\Rightarrow\; Arr(RP'+1) \neq sp \\
\quad \vee \\
\quad\quad (\!| RP+1 \,..\, RP' |\!) \cap \{sp, nl\} = \emptyset \\
\quad\quad RP' \neq Max \;\Rightarrow\; Arr(RP'+1) \in \{sp, nl\} \\
\quad rep! = \text{“OK”} \\
\vee \\
\quad RP = RP' = Max \\
\quad rep! = \text{“At bottom of docnment”}
\end{array}\right.
$$

Code for $RightDeleteWord_{Doc1}$

```
Standardize_Doc1 ;
{ LP = CP }
if
| (RP ≠ Max) -->
        if
        | (Arr(RP + 1) = nl) --> RP++
        □
        | (Arr(RP + 1) = sp) --> RDWSWord
        □
        | (Arr(RP + 1) ≠ nl ∧ Arr(RP + 1) ≠ sp) --> RDNWSWord
        fi ;
        return("OK")
□
| (RP = Max) --> return("At bottom of document")
fi
```

RDWSWord

do

$$
\begin{array}{|l}
(RP \neq Max \wedge Arr(RP+1) = sp) \longrightarrow RP{+}{+} \\
\;\{\textsf{Invariant}: Arr \, (\!| RP_o + 1 \,..\, RP |\!) \subseteq \{sp\}\} \\
\;\{\textsf{Variant}: Max - RP\} \\
\;\{\textsf{Guard Negation}: RP \neq Max \Rightarrow Arr(RP+1) \neq sp\}
\end{array}
$$

od

RDNWSWord

do

$$
\begin{array}{|l}
(RP \neq Max \wedge Arr(RP+1) \neq nl \wedge Arr(RP+1) \neq sp) \longrightarrow RP{+}{+} \\
\;\{\textsf{Invariant}: Arr \, (\!| RP_o + 1 \,..\, RP |\!) \cap \{sp, nl\} = \emptyset\} \\
\;\{\textsf{Variant}: Max - RP\} \\
\;\{\textsf{Guard Negation}: RP \neq Max \Rightarrow Arr(RP+1) \in \{sp, nl\}\}
\end{array}
$$

od

Notes

Simplification and refinement:

We apply FW to both sides of the first, second and fifth predicates of the weakest concrete operation and choose to leave the array unchanged - [$\sqsubseteq$ 2 : 3.2*a*] *Safety*.

Further Simplification:

We use Lemma 3:1.2.2a to simplify the second predicate of the first disjunct, noting that C and C' of that lemma correspond to Arr after RP and Arr after RP' respectively.

Code:

Again we use the *Standard* configuration, by appealing to Lemma 4:2.3.1a. We then use [$\sqsubseteq$ 2 : 3.5.2*b*] twice to give the two alternate constructs (the pre-condition of the disjuncts forming the guards in both cases). Finally, the two loops are justified by appealing to [$\sqsubseteq$ 2 : 3.5.3*a*]. Note that the negation of the loop guard is:

$$RP = Max \vee Arr(RP+1) \neq sp$$

which is logically equivalent to:

$$RP \neq Max \Rightarrow Arr(RP+1) \neq sp$$

and we frequently use this latter implication form to more easily demonstrate that the guard negation is equivalent to a predicate of the refinement.

2.4 Cursor-Changing Operations

As discussed in Section 2.1, a cursor-changing operation does not require a reconfiguration of the concrete state: the design decision enables such operations to be effected by a change of CP.

When specifying a cursor-changing operation in the abstract specification, we used

$$\begin{array}{|l}
\multicolumn{1}{l}{\Xi_{Cont} Doc1} \\
\Delta Doc1 \\
\hline
(Left_{Char} \frown Right_{Char}) = (Left_{Char}' \frown Right_{Char}') \\
\end{array}$$

and have, as the weakest concrete specification

$$\begin{array}{|l}
\multicolumn{1}{l}{\Xi_{Cont} Doc1\ \mathbf{C}} \\
\Delta ConcDoc1 \\
\hline
(ArrCont \text{ for } CP) \frown (ArrCont \text{ after } CP) = \\
\qquad (ArrCont' \text{ for } CP') \frown (ArrCont' \text{ after } CP') \\
\end{array}$$

The predicate part simplifies to:

$$\left[\ ArrCont = ArrCont'\right.$$

which [$\sqsubseteq 2 : 3.2a$] is refined by:

$$\left[\begin{array}{l} Arr' = Arr \\ RP' = RP \\ LP' = LP \end{array}\right.$$

and thus we may replace the former by the latter in the refinement process.

When refining such operations, values of *ArrCont* will be required (rather than those of *Arr* when using a *Standard* configuration), and we now specify an operation which will, for an input of *ptr*, output the contents of that location of *ArrCont*, *c*:

$$\begin{array}{|l}
\multicolumn{1}{l}{GetArrCont} \\
ConcDoc1 \\
ptr? : 1 \mathinner{\ldotp\ldotp} Max \\
c! : Char \\
\hline
c! = ArrCont\ ptr? \\
\end{array}$$

Code for *GetArrCont*(*ptr*) $\qquad ptr? : 1 \mathinner{\ldotp\ldotp} Max$

if
$\mid (ptr \leq LP) \longrightarrow return(Arr\ ptr)$
$\square$
$\mid (ptr > LP) \longrightarrow return(Arr\ (ptr + RP - LP))$
fi

Note

The code follows from [$\sqsubseteq 2 : 3.5.2b$] and Lemma 4:2.1.1d.

2.4.1 Refinement Of "LeftMoveLine"

Specification :

$$(LeftMvLine \wedge Success) \vee ErrorTopOfDoc$$

Expansion:

$LeftMoveLine_{Doc1}$

$\Xi_{Cont} Doc1$

$$Left_{Line} \neq << >>$$
$$Left_{Line}' = \text{front } Left_{Line}$$
$$rep! = \text{"OK"}$$
$$\vee$$
$$Left_{Line} = Left_{Line}' = << >>$$
$$rep! = \text{"At top of document"}$$

Weakest concrete operation:

$LeftMoveLine_{Doc1}$ **C**

$\Xi_{Cont} Doc1$ **C**

$$\mathsf{FL}^{-1}(ArrCont \text{ for } CP) \neq << >>$$
$$\mathsf{FL}^{-1}(ArrCont' \text{ for } CP') = \text{front } \mathsf{FL}^{-1}(ArrCont \text{ for } CP)$$
$$rep! = \text{"OK"}$$
$$\vee$$
$$\mathsf{FL}^{-1}(ArrCont \text{ for } CP) = \mathsf{FL}^{-1}(ArrCont' \text{ for } CP') = << >>$$
$$rep! = \text{"At top of document"}$$

Simplification and refinement:

$$
\left[\begin{array}{l}
Arr', LP', RP' = Arr, LP, RP \\
\quad CP' \leq CP \\
\quad\quad \textsf{last}\ (ArrCont\ \textsf{for}\ CP) = nl \\
\quad\quad CP' = CP - 1 \\
\quad \vee \\
\quad\quad nl \notin ArrCont\,(\!|\ CP' + 1 \mathinner{..} CP\ |\!) \\
\quad\quad CP' \neq 0 \;\Rightarrow\; ArrCont(CP') \neq nl \\
\quad rep! = \text{“OK”} \\
\vee \\
\quad CP = CP' = 0 \\
\quad rep! = \text{“At top of document”}
\end{array}\right.
$$

Code for $LeftMoveLine_{Doc1}$

if
  | $(CP \neq 0)$ —>
    if
      | $(GetArrCont(CP) = nl)$ —> $CP := CP - 1$
    □
      | $(GetArrCont(CP) \neq nl)$ —>
        do
          | $(CP \neq 0 \wedge GetArrCont(CP) \neq nl)$ —> CP--
          | { Invariant : $nl \notin ArrCont\,(\!|\ CP + 1 \mathinner{..} CP_o\ |\!)$ }
          | { Variant : CP }
          | { Guard Negation : $CP \neq 0 \Rightarrow ArrCont\,(CP) = nl$ }
        od
    fi ;
    $return$(“OK”)
□
  | $(CP = 0)$ —> $return$(“At top of document”)
fi
{ $CP \leq CP_o$ }

Notes

Simplification and refinement:

We incorporate the predicate of $\Xi_{Cont}Doc1$ **C** and apply FL to the first predicate of the first disjunct, and to the second disjunct. We simplify the second predicate of the first disjunct using Lemma 3:1.2.2b, noting that C and C' correspond to $ArrCont$ for CP and $ArrCont$ for CP' respectively.

Code:

We appeal to [$\sqsubseteq$ **2** : 3.5.2b] for both alternate constructs and [$\sqsubseteq$ 2 : 3.5.3a] for the loop.

3 Refinement Of Doc3

The *Doc2* state introduces an unbounded display of the editor, and the *Doc3* state imposes invariants on that model by ensuring that no line can end in whitespace (other than the current line, when the cursor is at its right hand end) and that the document cannot end in null lines (except when the current line is the last line). Since both abstract states are concerned with how the edited document will look on the terminal screen, we conduct refinement in a single step, on the *Doc3* state.

3.1 The Design Decision

The *Doc3* state does not extend (i.e. does not introduce new variables on) the *Doc2* state and so we first consider the concrete representation of the latter. It comprises the *UD* and *Doc1* states, and includes an invariant relationship between *UDLines*, *UDCurLine*, *UDCurX* and *UDCurY* of *UD*, and $Left_{Char}$ and $Right_{Char}$ of *Doc1*.

The invariant renders the *UDLines* and *UDCurLine* components redundant since both may be calculated from *Doc1* using *UDCurX* and *UDCurY* (Part 3, Section 2.2); we choose not to represent the *UDLines* sequence in the implementation. However, since changes take place in the current line, we do wish to have a representation for *UDCurLine*, which we provide by including two pointers, *StartIn* and *EndIn*, which point to the *ArrCont* location preceding the start of the cursor line, and the end of the cursor line, respectively. Both share the same signature, having a minimum value of *0* and a maximum value of *Max*.

We require a representation for both of the abstract variables *UDCurX* and *UDCurY* in the concrete state: we introduce *CurX* and *CurY*, the minimum value for each being *1* (when the cursor is at the top left of the document) and the maximum value (*Max* + *1*) (provided by the document containing no newline characters in the case of *UDCurX*, or containing only newline characters in the case of *UDCurY*). Thus *CurX* will always exceed the difference between *CP* and *StartIn* by one, and *CurY* will exceed the number of newlines in the *ArrCont* locations up to *CP* by one.

We also incorporate the variable *DocNL*, representing the number of newline characters in the document (and so being equal to one less that the number of lines in the document), principally for optimization of *MoveToBot* - obviating the need for a newline count to set *CurY* - but also to identify more easily those occasions when a cursor movement (*Doc4*) operation would move the cursor below the unbounded display of the document

Finally, we include the two variables *WSRem* and *NLRem*: since some *cursor* operations will result in a change in the content of the document (when trailing whitespace/null lines are removed) these variables will represent the amount of whitespace removed by *RemTrailWS* and null lines removed by *RemTrailNL*, thereby enabling the repositioning of the *Mark* pointer (Section 5) so that its same relative position in the document is maintained.

We therefore have, as our concrete representation of *Doc2*

ConcDoc2

$$
\begin{array}{l}
ConcDoc1 \\
StartIn, EndIn, DocNL, WSRem, NLRem : 0 \,..\, Max \\
CurX, CurY : 1 \,..\, Max + 1 \\
\hline
StartIn \leq CP \leq EndIn \\
\mathsf{NoNLin}\ (ArrCont, StartIn + 1 \,..\, EndIn) \\
StartIn \neq 0 \;\Rightarrow\; ArrCont\ StartIn = nl \\
EndIn \neq (Max + LP - RP) \;\Rightarrow\; ArrCont(EndIn + 1) = nl \\
CurX = CP - StartIn + 1 \\
CurY = \mathsf{NumNLin}\ (ArrCont, 1 \,..\, CP) + 1 \\
DocNL = \mathsf{TotalNLin}\ ArrCont
\end{array}
$$

where

$$
\begin{array}{l}
\mathsf{NumNLin} : (1 \,..\, Max \longrightarrow Char) \times \mathbb{P}\,\mathbb{N} \longrightarrow \mathbb{N} \\
\mathsf{TotalNLin} : (1 \,..\, Max \longrightarrow Char) \longrightarrow \mathbb{N} \\
\mathsf{NoNLin}\ \_ : (1 \,..\, Max \longrightarrow Char) \times \mathbb{P}\,\mathbb{N} \longrightarrow \mathbb{B} \\
\hline
\mathsf{NumNLin}\ (array, m \,..\, n) = \#((m \,..\, n \lhd array) \rhd \{nl\}) \\
\mathsf{TotalNLin}\ array = \mathsf{NumNLin}\ (array, 1 \,..\, \#\, array) \\
\mathsf{NoNLin}\ (array, m \,..\, n) \;\Leftrightarrow\; \mathsf{NumNLin}\ (array, m \,..\, n) = 0
\end{array}
$$

For each cursor-changing operation *OP* associated with the set *MoveOps* (Section 3:2.3), the document length will be changed only by the amount of whitespace/number of null lines removed, and therefore we wish the following invariant to hold after each such operation:

$$\{\, LP - RP = LP_o - RP_o + WSRem + NLRem \,\}$$

The following are a direct result of the *ConcDoc2* invariant:

Lemma 4 : 3.1a

$$
\begin{array}{l}
\quad ConcDoc2 \\
\vdash \\
\quad CurY = \mathsf{NumNLin}\ (ArrCont, 1 \,..\, CP) + 1 \\
\quad CurY = \mathsf{NumNLin}\ (ArrCont, 1 \,..\, StartIn) + 1 \\
\quad CurY = \mathsf{NumNLin}\ (ArrCont, 1 \,..\, EndIn) + 1 \\
\blacksquare
\end{array}
$$

Proof

Follows since there are no newlines in the *ArrCont* locations from (*Startln*+ *1*) to *Endln*, and since *CP* must lie between *Startln* and *Endln*.

■

Corollary 4 : 3.1b

$$ConcDoc2$$

$$\vdash$$

$$CurY = \#(\mathsf{FDL}^{-1}(ArrCont \text{ for } CP))$$
$$CurY = \#(\mathsf{FDL}^{-1}(ArrCont \text{ for } Startln))$$
$$CurY = \#(\mathsf{FDL}^{-1}(ArrCont \text{ for } Endln))$$

■

Proof

Follows from the previous lemma and Lemma 3:2.2b.

■

Lemma 4 : 3.1c

$$ConcDoc2 \mid CP = 0$$

$$\vdash$$

$$CurX = CurY = Startln + 1 = 1$$

■

Proof

Since *Startln* may not exceed *CP*, *Startln* must also be zero, and so *CurX* is unity (since it exceeds the difference between *CP* and *Startln* by one), as is *CurY* (since it is one more than the number of newlines in the range (*1* .. *CP*), which is empty).

■

Lemma 4 : 3.1d

$$ConcDoc2 \mid CP = Max + LP - RP$$

$$\vdash$$

$$Endln = Max + LP - RP \quad \wedge \quad CurY = DocNL + 1$$

■

Proof

Since *CP* may not exceed *Endln*, (*Max* + *LP* − *RP*) is the maximum value of *CP*, and so *Endln* must also have that value. *DocY* is one more than the number of newlines up to *CP* - i.e. the number of newlines in *ArrCont* - and so exceeds *DocNL* by one.

■

Lemma 4 : 3.1e

$$ConcDoc2 \mid CP \neq 0 \wedge ArrCont\ CP = nl$$
$$\vdash$$
$$StartIn = CP \wedge CurX = 1$$

■

Proof

Since CP must be in the non-empty range $(StartIn \,..\, EndIn)$, and there are no newlines in the $ArrCont$ locations $(StartIn + 1 \,..\, EndIn)$, CP must equal $StartIn$, and $CurX$, exceeding their difference by one, must be unity.

■

Lemma 4 : 3.1f

$$ConcDoc2 \mid CP \neq Max + LP - RP \wedge ArrCont(CP + 1) = nl$$
$$\vdash$$
$$EndIn = CP \wedge CurX = EndIn - StartIn + 1$$

■

Proof

Similar to Lemma 4:3.1e.

■

We now impose the whitespace and null lines invariants on the $ConcDoc2$ model to give:

ConcDoc3

ConcDoc2
$\forall\, i : 1 \,..\, DocNL + 1 - \{CurY\} \bullet \mathsf{visible}\ ((\mathsf{FDL}^{-1}\ ArrCont)\ i)$ $\mathsf{visible}\ (CP + 1 \,..\, EndIn \lhd ArrCont)$ $\mathsf{visibleseq}\ (\mathsf{FDL}^{-1}(ArrCont\ \mathsf{after}\ EndIn))$

3.1.1 The Concrete-Abstract Invariant

$$
\begin{array}{|l}
\textit{Rel}_{Doc3} \\
\hline
\textit{Doc3} \\
\textit{ConcDoc3} \\
\textit{Rel}_{Doc1} \\
\hline
\textit{UDCurX}, \textit{UDCurY} = \textit{CurX}, \textit{CurY} \\
\textit{UDLines} = \mathsf{FDL}^{-1}\ \textit{ArrCont} \\
\textit{UDCurLine} = \textit{StartIn} + 1 \mathinner{..} \textit{EndIn} \upharpoonleft \textit{ArrCont} \\
\#\ \textit{UDLines} = \textit{DocNL} + 1 \\
\hline
\end{array}
$$

The following are direct results of this schema:

Lemma 4 : 3.1.1a

$$\textit{Rel}_{Doc3}$$
$$\vdash$$
$$\textit{UDCurLine} = \langle\,\rangle \quad \Leftrightarrow \quad \textit{StartIn} = \textit{CP} = \textit{EndIn}$$

■

Lemma 4 : 3.1.1b

$$\textit{Rel}_{Doc3}$$
$$\vdash$$
$$\textit{UDCurLine}\ \mathsf{for}\ \textit{UDCurX} - 1 \quad = \quad \textit{StartIn} + 1 \mathinner{..} \textit{CP} \upharpoonleft \textit{ArrCont}$$
$$\textit{UDCurLine}\ \mathsf{after}\ \textit{UDCurX} - 1 \quad = \quad \textit{CP} + 1 \mathinner{..} \textit{EndIn} \upharpoonleft \textit{ArrCont}$$

■

In order to discharge our data refinement proof obligations we must show that the concrete state defines a valid abstract state. $\textit{Rel}_{Doc3}$ ensures that each of the abstract components is of the correct type, and is uniquely defined. We must show that the abstract invariant is satisfied.

Firstly, the first result of Lemma 4:3.1b implies that ($\textit{CurY} \leq \#\mathsf{FDL}^{-1}\ \textit{ArrCont}$), and so $\textit{Rel}_{Doc3}$ ensures that ($\textit{UDCurY} \leq \#\ \textit{UDLines}$).

The invariant of *ConcDoc2* ensures that *ArrCont* in the range $\textit{StartIn} + 1 \mathinner{..} \textit{EndIn}$ is a member of the display line sequence corresponding to *ArrCont*, $\mathsf{FDL}^{-1}\ \textit{ArrCont}$; Lemma 4:3.1b ensures that it is the $\textit{CurY}^{th}$ member of the sequence and, by $\textit{Rel}_{Doc3}$, *UDCurLine* is the $\textit{UDCurY}^{th}$ member of *UDLines*.

Finally, *ConcDoc3* defines that *CurX*, and hence *UDCurX*, cannot exceed ($\textit{EndIn} - \textit{StartIn} + 1$), which, by $\textit{Rel}_{Doc3}$, is $\#\ \textit{UDCurLine}+1$.

Finally, *ConcDoc3* imposes the whitespace and null lines invariants on *ConcDoc2* in exactly the same way the *Doc3* imposes them on *Doc2*, and Lemma 4:3.3.3b ensures that the concrete state invariant matches that of the abstract state.

Thus the *Doc3* invariant is satisfied, and we establish:

Lemma 4 : 3.1.1c

$$Rel_{Doc3}$$
$$\vdash$$
$$\forall\ ConcDoc3 \bullet \exists_1\ Doc3 \bullet Rel_{Doc3}$$

■

We now calculate $AbsRel_{Doc3}$ and have, after simplification:

$$AbsRel_{Doc3} \mathrel{\hat=} \Xi Doc3 \land AbsRel_{Doc1}$$

and, pursuing the same argument as in Section 2.1.1, we obtain:

Lemma 4 : 3.1.1d

$$\vdash$$
$$Doc3 \sqsubseteq_{\propto} ConcDoc3$$

■

ConcDoc2 requires no specific configuration for the execution of its operations, but since we will be dealing with whitespace/newline removal for *ConcDoc3* operations, we define the *Standard* concrete state as:

$$ConcDoc3_{Standard} \mathrel{\hat=} ConcDoc3 \land ConcDoc1_{Standard}$$

and since *Doc3* contains *Doc1* and Rel_{Doc3} contains Rel_{Doc1}, we have:

$$Rel_{Doc3\,Standard} \mathrel{\hat=} Rel_{Doc3} \land Rel_{Doc1\,Standard}$$

and we calculate $ConcRel_{Doc3\,Standard}$ which simplifies to give:

$ConcRel_{Doc3\,Standard}$

$$\Delta ConcDoc3$$

$$
\begin{aligned}
&CP', LP' = CP, CP\\
&ArrCont' = ArrCont\\
&StartIn', EndIn' = StartIn, EndIn\\
&CurX', CurY' = CurX, CurY\\
&DocNL' = DocNL\\
&WSRem', NLRem' = WSRem, NLRem
\end{aligned}
$$

Hence a concrete reorganizing operation must preserve the newly-introduced components of the *Doc2* state; we may appeal to [$\sqsubseteq$ 2 : 3.2a] to show that $Standardize_{Doc1}$ refines $ConcRel_{Doc3}$:

$$ConcRel_{Doc3\,Standard} \;\sqsubseteq\; Standardize_{Doc1}$$

3.2 Initialization

Specification :

$$Initialize_{Doc1} \;\wedge\; \Delta Doc2$$

Codefor $Initialize_{Doc3}$

$Initialize_{Doc1}$; $StartIn := 0$; $EndIn := 0$;
$CurX := 1$; $CurY := 1$; $DocNL := 0$;
$WSRem := 0$; $NLRem := 0$

Note
We use the results of Lemma 3:2.2d, 4:3.1.1a and 4:3.1.1c; we set *WSRem* and *NLRem* to zero (although they could have any values).

3.3 Operations To Set "StartIn" And "EndIn"

For each content-changing operation, and cursor-changing operation in which the cursor line is changed, it will be necessary to set new values for *StartIn* and/or *EndIn*.

We specify and refine an operation, *SetEndIn* that sets *EndIn* using the (new) cursor position: an analogous operation, *SetStartIn* may be similarly specified and refined.

Specification:

SetEndIn

$EndIn' : 0 \,..\, Max$
ConcDoc2

$EndIn' \neq Max + LP - RP \;\Rightarrow\; ArrCont\ (EndIn' + 1) = nl$
$\textsf{NoNLin}\ (ArrCont, CP + 1 \,..\, EndIn')$

Code for *SetEndln*

$EndIn := CP$;
do
$(EndIn < Max + LP - RP \wedge GetArrCont(EndIn + 1) \neq nl) \longrightarrow EndIn{+}{+}$
{ Invariant : NoNLin $(ArrCont, CP_o + 1 \mathrel{..} EndIn)$ }
{ Invariant : $CP \leq EndIn$ }
{ Variant : $Max + LP_o - RP_o - EndIn$ }
{ Guard Negation : $EndIn \neq Max + LP_o - RP_o \Rightarrow ArrCont(EndIn + 1) = nl$ }
od

Note

The second invariant of *ConcDoc2* may be equivalently stated:

NoNLin $(ArrCont, StartIn + 1 \mathrel{..} CP)$
NoNLin $(ArrCont, CP + 1 \mathrel{..} EndIn)$

The second is satisfied by this operation, the first being satisfied by *SetStartln*. Since we are not changing the document's content we do not need to standardize the array, and we use *GetArrCont* to search *ArrCont*. We appeal to [$\sqsubseteq$ 3 : 5.3*a*] to refine to a loop.

3.3.1 An Operation To Update The ConcDoc2 Components

Each abstract operation *OP* defined on the *Doc1* state is promoted to the *Doc2* state in the same way:

$$OP_{Doc2} \mathrel{\widehat{=}} OP_{Doc1} \wedge \Delta Doc2$$

to give, as the weakest specification of the concrete operation:

$$OP_{Doc2}\mathbf{C} \mathrel{\widehat{=}} OP_{Doc1}\mathbf{C} \wedge \Delta ConcDoc2$$

An intuitive approach suggests that only the new components of the *Doc2* state - *CurX*, *CurY*, *Startln*, *Endln* and *DocNL* - need be considered, since the refinement on *Doc1* will already have dealt with the other components. This, in fact, is the way we proceed: we specify and refine a concrete operation that updates the *ConcDoc2* components, $Update_{Doc2}$, and show that post-sequential composition of the operation with OP_{Doc1} refines both OP_{Doc1} and the promotion schema $\Delta Doc2$, and, using [$\sqsubseteq$ 2 : 3.2*f*], OP_{Doc2}:

Lemma 4 : 3.3.1a

$$OP_{Doc1} \sqsubseteq OP_{Doc1} \; ; \; Update_{Doc2}$$
$$\Delta Doc2 \sqsubseteq OP_{Doc1} \; ; \; Update_{Doc2}$$

$\vdash$

$$OP_{Doc2} \mathrel{\widehat=} OP_{Doc1} \land \Delta Doc2 \sqsubseteq OP_{Doc1} \; ; \; Update_{Doc2}$$

■

We note that if the update operation is total and does not change any of the *ConcDoc1* components, the first antecedent is automatically satisfied by appealing to [$\sqsubseteq$ 2 : 3.4*b*]. Therefore the above lemma will apply when the second antecedent is satisfied (since OP_{Doc1} will ensure the after-state satisfies the invariant for *Doc1*) - i.e. when $Update_{Doc2}$ produces an after-state satisfying the *Doc2* invariant. [$\sqsubseteq$ 2 : 3.4*c*] ensures that we may use the refinements of the *Doc1* operations derived in Section 2.

We may update *StartIn* and *EndIn* using the operations *SetStartIn* and *SetEndIn*, and then update the value of *CurX* using the *ConcDoc2* invariant relationship that it exceeds the difference between *CP* and *StartIn* by one.

Changes in *CurY* and/or *DocNL* will depend upon the type of operation, and we summarize the effect of each operation type on *CurY* and *DocNL*, where each is incremented (for *Right* operations) or decremented (for *Left* operations) by the number of newlines in the *ArrCont* or *Arr* locations indicated:

Operation Type	Component Affected	*ArrCont* locations	*Arr* locations
LeftInsert	*CurY*, *DocNL*	$CP + 1 \,..\, CP'$	
RightInsert	*DocNL*		$RP' + 1 \,..\, LP' - LP + RP$
LeftDelete	*CurY*, *DocNL*	$CP' + 1 \,..\, CP$	
RightDelete	*DocNL*		$LP' - LP + RP \,..\, RP'$
LeftMove	*CurY*	$CP' + 1 \,..\, CP$	
RightMove	*CurY*	$CP + 1 \,..\, CP'$	

Note that since insert and delete operations will have been effected on a standard configuration by a change of pointer, the "deleted" characters still reside in the array available for inspection. In the case of a right insert or delete, however, we calculate the previous standardized position of *RP* (rather than using *prevRP*), since we may not assume a standardized configuration *before* the commencement of the *Doc1* operation.

We now give the operation $Update_{Doc2}$:

Code for $Update_{Doc2}(prevCP, prevLP, prevRP)$ $\qquad prevCP?, prevLP?, prevRP? : 0 \,..\, Max$

```
NumNL : 0 .. Max ;
{ CurY = NumNLin (ArrCont, 1 .. prevCP) + 1 }
if
| (OPType ≠ NoMove)   -->  SetStartln ; SetEndln ; CurX := (CP − Startln + 1) ;
|                          WSRem, NLRem := 0, 0 ; Update_OPType
[]
| (OPType = NoMove)   -->  Skip
fi
```

Update_OPType

```
if
| (OPType = LeftInsert)  -->  Update_LeftInsert_Doc2
[]
| (OPType = RightInsert) -->  Update_RightInsert_Doc2
[]
| (OPType = LeftDelete)  -->  Update_LeftDelete_Doc2
[]
| (OPType = RightDelete) -->  Update_RightDelete_Doc2
[]
| (OPType = LeftMove)    -->  Update_LeftMove_Doc2
[]
| (OPType = RightMove)   -->  Update_RightMove_Doc2
fi
```

$Update\_LeftInsert_{Doc2}$

```
{ LP = CP }
{ CurY = NumNLin (ArrCont, 1 .. prevCP) + 1 }
{ DocNL = NumNLin (ArrCont, 1 .. prevCP) + NumNLin (ArrCont, CP + 1 .. Max) }
NumNL := NLCountArr(prevCP, CP) ;
{ NumNL = NumNLin (ArrCont, prevCP + 1 .. CP) }
CurY := CurY + NumNL ; DocNL := DocNL + NumNL
{ CurY = NumNLin (ArrCont, 1 .. CP) + 1 }
{ DocNL = NumNLin (ArrCont, 1 .. Max) = TotalNLin ArrCont }
```

$NLCountArr(first, last) \qquad first?, last? : 0 \,..\, Max$

$NumNL := 0 : 0 \,..\, Max$;

do

| $(first \neq last) \longrightarrow$

$first$++ ;

if

| $(Arr\ first = nl) \longrightarrow NumNL$++

$\Box$

| $(Arr\ first \neq nl) \longrightarrow Skip$

fi

{ Invariant : $NumNL =$ NumNLin $(Arr.first_o + 1 \,..\, first)$ }

{ Variant : $last - first$ }

{ Guard Negation : $first = last$ }

od ;

{ $NumNL =$ NumNLin $(Arr.first_o + 1 \,..\, last)$ }

$return(NumNL)$

Notes

We wish to show that $Update_{Doc2}$ refines $\Delta Doc2$ by ensuring that the *ConcDoc2* components are set in line with the invariant. We use [$\sqsubseteq$ 2 : 3.5.2*a*] to obtain the first alternate construct (noting that the pre-condition of the first body is dictated by *Update_OPType* which is the same as the guard, and that *Skip* is total). We appeal to the result again to partition the *Update* operations, and we give the code for one; the code for the remaining five is similar. Its pre-condition is dictated by that of *NLCountArr* which requires that its second parameter is not less than its first, and this is ensured by the guard. We use the results of Lemmas 4:3.4.1d and 4:3.4.1e to save unnecessary counting of newlines when the cursor is moved either to the top or to bottom of the document, and appeal to [$\sqsubseteq$ 2 : 3.5.3*a*] for the loop of *NLCountArr*.

If an operation takes input parameters, we include them in brackets after the operation name; we also list the parameters decorated with '?' and giving their signatures so that they may be easily identified in the earlier development.

3.4 The Removal Of Trailing Whitespace

Having refined the operation to the *Doc2* state, our next task is the removal of trailing whitespace, since the abstract promotion method to the *Doc3* state is by pre sequential composition with *FlagPrevCursor* and then post sequential composition with *RemTrailWS* (followed by sequential composition with *RemTrailNL*), where

$$FlagPrevCursor \ \widehat{=}\ [\ Left_{Char} : seq\ Char\ ;\ prevCP! : \mathbb{N} \mid prevCP! = \#\, Left_{Char}\ ]$$

$RemTrailWS$

$$\Delta UD$$
$$prevCP? : \mathbb{N}$$

$$UDCurX', UDCurY' = UDCurX, UDCurY$$
$$\{prevUDCurY\} \ntriangleleft UDLines' = \{prevUDCurY\} \ntriangleleft UDLines$$
$$\quad prevUDCurY = UDCurY$$
$$\quad UDCurLine' \text{ for } UDCurX - 1 = UDCurLine \text{ for } UDCurX - 1$$
$$\quad (UDCurLine' \text{ after } UDCurX - 1) \text{ visible\_prefix } (UDCurLine \text{ after } UDCurX - 1)$$
$$\vee$$
$$\quad prevUDCurY \neq UDCurY$$
$$\quad (UDLines' \ prevUDCurY) \text{ visible\_prefix } (UDLines \ prevUDCurY)$$
$$\text{where}$$
$$\quad prevUDCurY = \#(\mathsf{FDL}^{-1}((Left_{Char} \frown Right_{Char}) \text{ for } prevCP?))$$

The weakest concrete operation for the latter is:

$$CurX', CurY' = CurX, CurY$$
$$\{prevCurY\} \ntriangleleft (\mathsf{FDL}^{-1} ArrCont') = \{prevCurY\} \ntriangleleft (\mathsf{FDL}^{-1} ArrCont)$$
$$\quad prevCurY = CurY$$
$$\quad (StartIn' + 1 \mathinner{..} EndIn' \upharpoonleft ArrCont') \text{ for } CurX - 1 =$$
$$\qquad (StartIn + 1 \mathinner{..} EndIn \upharpoonleft ArrCont) \text{ for } CurX - 1$$
$$\quad ((StartIn' + 1 \mathinner{..} EndIn' \upharpoonleft ArrCont') \text{ after } CurX - 1) \text{ visible\_prefix}$$
$$\qquad ((StartIn + 1 \mathinner{..} EndIn \upharpoonleft ArrCont) \text{ after } CurX - 1)$$
$$\vee$$
$$\quad prevCurY \neq CurY$$
$$\quad ((\mathsf{FDL}^{-1} ArrCont') \ prevCurY) \text{ visible\_prefix } ((\mathsf{FDL}^{-1} ArrCont) \ prevCurY)$$
$$\text{where}$$
$$\quad prevCurY = \mathsf{NumNLin}\ (ArrCont, 1 \mathinner{..} prevCP?) + 1$$

Simplification:

$$
\left[\begin{array}{l}
CurX', CurY' = CurX, CurY \\
\quad prevCurY = CurY \\
\quad ArrCont' \text{ for } StartIn = ArrCont \text{ for } StartIn \\
\quad ArrCont' \text{ after } EndIn' = ArrCont \text{ after } EndIn \\
\quad (StartIn + 1 \mathinner{..} CP \upharpoonleft ArrCont') = (StartIn + 1 \mathinner{..} CP \upharpoonleft ArrCont) \\
\quad (CP + 1 \mathinner{..} EndIn' \upharpoonleft ArrCont') \text{ visible\_prefix } (CP + 1 \mathinner{..} EndIn \upharpoonleft ArrCont) \\
\lor \\
\quad prevCurY \neq CurY \\
\quad ArrCont' \text{ for } prevStartIn = ArrCont \text{ for } prevStartIn \\
\quad ArrCont' \text{ after } newEndIn = ArrCont \text{ after } prevEndIn \\
\quad (prevStartIn + 1 \mathinner{..} newEndIn \upharpoonleft ArrCont') \text{ visible\_prefix} \\
\qquad\qquad\qquad\qquad\qquad\qquad (prevStartIn + 1 \mathinner{..} prevEndIn \upharpoonleft ArrCont) \\
\text{where} \\
\quad prevStartIn \leq newEndIn \leq prevEndIn \\
\quad prevStartIn \leq prevCP? \leq prevEndIn \\
\quad \mathsf{NoNLin}\ (ArrCont, prevStartIn + 1 \mathinner{..} prevEndIn) \\
\quad prevStartIn \neq 0 \;\Rightarrow\; ArrCont\ prevStartIn = nl \\
\quad prevEndIn \neq Max + LP - RP \;\Rightarrow\; ArrCont(prevEndIn + 1) = nl \\
\quad prevCurY = \mathsf{NumNLin}\ (ArrCont, 1 \mathinner{..} prevEndIn) + 1
\end{array}\right.
$$

Further simplification and refinement:

$$
\left[\begin{array}{l}
CurX', CurY' = CurX, CurY \\
\quad prevCurY = CurY \\
\quad ArrCont' = ArrCont \text{ for } EndIn' \frown ArrCont \text{ after } EndIn \\
\quad ArrCont \llparenthesis EndIn' + 1 \mathinner{..} EndIn \rrparenthesis \subseteq \{sp\} \\
\quad EndIn' \neq CP \;\Rightarrow\; ArrCont'\ EndIn' \neq sp \\
\lor \\
\quad prevCurY \neq CurY \\
\quad ArrCont' = ArrCont \text{ for } newEndIn \frown ArrCont \text{ after } prevEndIn \\
\quad ArrCont \llparenthesis newEndIn + 1 \mathinner{..} prevEndIn \rrparenthesis \subseteq \{sp\} \\
\quad newEndIn \neq 0 \;\Rightarrow\; ArrCont'\ newEndIn \neq sp \\
\text{where} \\
\quad newEndIn \leq prevEndIn \\
\quad prevCP? \leq prevEndIn \\
\quad prevEndIn \neq Max + LP - RP \;\Rightarrow\; ArrCont(prevEndIn + 1) = nl \\
\quad \mathsf{NoNLin}\ (ArrCont, prevCP? + 1 \mathinner{..} prevEndIn) \\
\quad prevCurY = \mathsf{NumNLin}\ (ArrCont, 1 \mathinner{..} prevEndIn) + 1
\end{array}\right.
$$

Code for *RemTrailWS(prevCP)* $\quad$ *prevCP? : 0 .. Max*

```
tempCP := CP : 0 .. Max ;
tempEndln := Endln : 0 .. Max ;
prevEndln : 0 .. Max ;
GetprevEndln ;
if
| (OPType = LeftMove ∨ OPType = RightMove ∨
|  OPType = LeftInsert ∨ OPType = RightInsert) ⟶
|         if
|         | (prevEndln = Endln) ⟶ RemTrailWSUDCurLine
|         □
|         | (prevEndln ≠ Endln) ⟶ RemAllWSPrevUDCurLine
|         fi
□
| ¬(OPType = LeftMove ∨ OPType = RightMove ∨
|    OPType = LeftInsert ∨ OPType = RightInsert) ⟶ Skip
fi
```

GetprevEndln

```
CP := prevCP ; SetEndln ; prevEndln := Endln ;
CP := tempCP ; Endln := tempEndln
{ prevCP ≤ prevEndln }
{ Endln ≠ Max + LP − RP ⇒ ArrCont(Endln + 1) = nl }
{ NoNLin (ArrCont, prevCP + 1 .. prevEndln }
```

RemTrailWSUDCurLine

if

| $(EndIn \neq CP \wedge GetArrCont\ EndIn = sp) \longrightarrow$

$CP := EndIn\ ;\ Standardize_{Doc1}\ ;\ CP := tempCP\ ;$

$\{\ ArrCont = ArrCont_o \wedge CP = CP_o \wedge LP = EndIn_o\ \}$

do

| $(EndIn \neq CP \wedge Arr\ EndIn = sp) \longrightarrow EndIn--\ ;\ LP--\ ;\ WSRem++\ ;$

$\{$ Invariant : $ArrCont' = ArrCont$ for $EndIn \frown ArrCont$ after $EndIn_o\ \}$

$\{$ Invariant : $ArrCont\ (\!|\ EndIn + 1\ ..\ EndIn_o\ |\!)\ \subseteq\ \{sp\}\ \}$

$\{$ Invariant : $LP_o - RP_o = LP - RP + WSRem\ \}$

$\{$ Variant : $CP - EndIn\ \}$

$\{$ Guard Negation : $EndIn \neq CP \Rightarrow ArrCont\ EndIn \neq sp\ \}$

od

$\square$

| $\neg\,(EndIn \neq CP \wedge GetArrCont\ EndIn = sp) \longrightarrow Skip$

fi

RemAllWSPrevUDCurLine

if

$(prevEndln \neq 0 \ \wedge \ GetArrCont\ prevEndln = sp) \longrightarrow$

$CP := prevEndln\,;\ Standardize_{Doc1}\,;\ CP := tempCP\,;$

$\{\ ArrCont = ArrCont_o \ \wedge \ CP = CP_o \ \wedge \ LP = prevEndln\ \}$

do

$(LP \neq 0 \ \wedge \ Arr\ LP = sp) \longrightarrow LP--\,;\ WSRem++$

$\{\ \text{Invariant}: \ ArrCont' = ArrCont \text{ for } LP \frown ArrCont \text{ after } prevEndln\ \}$

$\{\ \text{Invariant}: \ ArrCont\ (\!|\ LP + 1 \,..\, prevEndln\ |\!) \ \subseteq \ \{sp\}\ \}$

$\{\ \text{Invariant}: \ LP_o - RP_o = LP - RP + WSRem\ \}$

$\{\ \text{Variant}: \ LP\ \}$

$\{\ \text{Guard Negation}: \ LP \neq 0 \ \Rightarrow \ ArrCont\ LP \neq sp\ \}$

od ;

$\{\ newEndln = LP\ \}$

if

$(CP > LP) \longrightarrow CP := (CP - WSRem)\,;$

$Startln := (Startln - WSRem)\,;$

$Endln := (Endln - WSRem)$

$\Box$

$\neg(CP > LP) \longrightarrow Skip$

fi

$\Box$

$\neg(prevEndln \neq 0 \ \wedge \ GetArrCont\ prevEndln = sp) \longrightarrow Skip$

fi

Notes:

Simplification and refinement of weakest concrete operation:

We use Lemma 4:3.1.1b to simplify the weakest concrete operation. Then in the first disjunct we use Lemma 3:3.2a (which states that a visible sequence reflexively satisfies the visible_prefix relation) together with Lemma 4:3.1.1b to provide the simplification; the second is similarly simplified, except that we must state the relationship between *prevCurY* and *prevEndln* (whereas they were part of the *Doc3* invariant in the first disjunct), and we choose to refine the operation by introducing the second of these as an input variable (thereby avoiding the needless re-calculation of the value *prevEndln*).

Code:

The code for *GetprevEndln* sets the *prevEndln* of the where clause.

We note that a right delete operation cannot result in the need for the removal of trailing whitespace, since if the cursor is not currently at the end of the line, the cursor position becomes the end of line, with the *ConcDoc3* invariant implying that there can be no trailing whitespace, and if the cursor is currently at the end of the line, the character that is moved to the end of the line as a result of the delete operation must currently be at the end of another document line, and since the *ConcDoc3* invariant holds before the operation it cannot be a space character.

Similarly a left delete operation does not require the removal of trailing whitespace since the character currently at the end of the cursor line will remain there, and, for the same reason as above, cannot be a space character.

If the previous cursor line is above the new cursor line, it is necessary to decrement *CP*, *StartIn* and *EndIn* by *WSRem* to ensure that each points to the same position of *ArrCont* as it did before the operation.

We refinement to the first alternate construct by [$\sqsubseteq$ 2 : 3.3*a*]; one guard is the negation of the other which satisfies *Domain*. We similarly refine to the second alternate construct (to *RemTrailWSUDCurLine* and *RemAllWSPrevUDCurLine*). We appeal to [$\sqsubseteq$ 2 : 3.5.3*a*] for the loop refinements, to provide the *Safety* aspect.

Although the operation requires a *Standard* configuration (for which we may use [$\sqsubseteq$ 4 : 2.3.1*a*]), we push the *Standardize* operation through until just before the loop is activated, otherwise the advantage of a non-*Standard* configuration for cursor-changing operations not requiring whitespace removal would be lost.

3.5 The Removal Of Trailing Null Lines

Having removed trailing whitespace, the final part of the abstract promotion method for *Doc2* operations to the *Doc3* state comprises sequential composition with *RemTrailNL*, with specification:

RemTrailNL

$\Delta Doc2$

$Left_{Line}' = Left_{Line}$
$Right_{Line}' \text{ visibleseq\_prefix } Right_{Line}$

Weakest concrete operation:

$$\left[\begin{array}{l} \mathsf{FDL}^{-1}(ArrCont' \text{ for } CP') = \mathsf{FDL}^{-1}(ArrCont \text{ for } CP) \\ (\mathsf{FDL}^{-1}(ArrCont' \text{ after } CP')) \text{ visibleseq\_prefix } (\mathsf{FDL}^{-1}(ArrCont \text{ after } CP)) \end{array}\right.$$

Simplification and refinement:

$$\left[\begin{array}{l} CP' = CP \\ ArrCont' = ArrCont \text{ for } Max + LP' - RP' \\ CP' \neq Max + LP' - RP' \Rightarrow ArrCont'(Max + LP' - RP') \neq nl \\ \mathbf{rng}\,(ArrCont \text{ after } Max + LP' - RP') \subseteq \{nl\} \\ Arr' = Arr \end{array}\right.$$

Code for *RemTrailNL*

$$
\begin{array}{l}
tempCP \ := \ CP : 0 \,..\, Max \ ; \\
\textbf{if} \\
\quad (OPType = LeftMove \ \vee \ OPType = RightInsert) \longrightarrow \\
\qquad \textbf{if} \\
\qquad\quad (CP \neq Max + LP - RP \ \wedge \ GetArrCont(Max + LP - RP = nl)) \longrightarrow \\
\qquad\qquad\quad CP \ := \ (Max + LP - CP) \ ; \ Standardize_{Doc1} \ ; \ CP \ := \ tempCP \ ; \\
\qquad\qquad\quad \{ \ ArrCont = ArrCont_o \ \wedge \ CP = CP_o \ \} \\
\qquad\qquad\quad \textbf{do} \\
\qquad\qquad\qquad (LP \neq CP \ \wedge \ Arr \ LP = nl) \longrightarrow \ LP{-}{-} \ ; \ DocNL{-}{-} \ ; \ NLRem{+}{+} \\
\qquad\qquad\qquad \{ \ \mathsf{Invariant:} \ LP = Max + LP - RP \ \} \\
\qquad\qquad\qquad \{ \ \mathsf{Invariant:} \ \mathsf{rng} \ (ArrCont_o \ \mathsf{after} \ LP) \ \subseteq \ \{nl\} \ \} \\
\qquad\qquad\qquad \{ \ \mathsf{Invariant:} \ LP_o - RP_o = LP - RP + NLRem \ \} \\
\qquad\qquad\qquad \{ \ \mathsf{Variant:} \ LP \ \} \\
\qquad\qquad\qquad \{ \ \mathsf{Guard\ Negation:} \ LP = CP \ \vee \ ArrCont \ LP \neq nl \ \} \\
\qquad\qquad\quad \textbf{od} \\
\qquad \square \\
\qquad\quad \neg(CP \neq Max + LP - RP \ \wedge \ GetArrCont(Max + LP - RP = nl)) \longrightarrow \ Skip \\
\qquad \textbf{fi} \\
\square \\
\quad \neg(OPType = LeftMove \ \vee \ OPType = RightInsert) \longrightarrow \ Skip \\
\textbf{fi}
\end{array}
$$

Notes

Simplification and refinement:

We apply FDL to both sides of the first predicate of the weakest concrete operation and so both *CP* and *ArrCont* up to *CP* will not change. We then use Lemma 3:3.2b: the second predicate is a direct result of *ArrCont'* after *CP* being a prefix of *ArrCont* after *CP* (and *ArrCont'* having length $CP + LP' - RP'$). We choose to leave the entire array unchanged.

Code:

We note that the only operation types that can result in the need for the removal of trailing null lines are a left move or right insert. Similar comments to those made for *RemTrailWS* regarding the refinement to alternate and loop constructs also apply here.

3.6 Promotion Of The Doc1 Operations

The abstract promotion method for each *OP* defined on the *Doc1* state to the *Doc3* state is:

$$FlagPrevCursor \ ; \ (OP_{Doc1} \ \wedge \ \Delta Doc2) \ ; \ RemTrailWS \ ; \ RemTrailNL$$

We have shown (Lemma 4:3.3.1a) that

$$OP_{Doc1} \wedge \Delta Doc2 \sqsubseteq OP_{Doc1} \; ; \; Update_{Doc2}$$

and we define:

$$Update_{Doc3} \;\widehat{=}\; Update_{Doc2}(prevCP, prevLP, prevRP) \; ; \; RemTrailWS(prevCP) \; ; \; RemTrailNL$$

and appeal to [$\sqsubseteq$ 2 : 3.4e] to give, for each concrete operation OP defined on the *ConcDoc1* state:

Code for OP_{Doc3}

$rep : Report$;
$FlagCPLPRP$;
$(rep := OP_{Doc1})$; $Update_{Doc3}(prevCP, prevLP, prevRP)$; $return(rep)$

4 Refinement Of Doc4

The abstract QP enables cursor movement outside the unbounded display of the document, and because of this we choose to conduct refinement on this state.

4.1 The Design Decision

The QP state comprises a cursor position represented by *QPCurX* and *QPCurY* and the *Doc4* invariant renders it redundant (since it is the same as the *Doc* cursor). We therefore choose to exclude the QP cursor components from the design decision.

However, for the same reasons as those for including *WSRem* and *NLRem* in *ConcDoc3*, we here include the analogous *WSIns* and *NLIns*, representing the amount of whitespace/number of newlines introduced by *PadWSNL*:

$$ConcDoc4 \;\widehat{=}\; ConcDoc3 \; ; \; WSIns, NLIns : 0 \,..\, Max$$

For each cursor-changing operation in the set *EditOps1* (Section 3:2.3) or the set *QPCursorOps* (Section 3:4.2.2), therefore, we wish the following to hold

$$\{ \; LP_o - RP_o = LP - RP + WSRem + NLRem - WSIns - NLIns \; \}$$

4.1.1 The Concrete-Abstract Invariant

$$\begin{array}{|l}
\multicolumn{1}{l}{Rel_{Doc4}} \\
Doc4 \\
ConcDoc4 \\
Rel_{Doc3} \\
\hline
QPCurX, QPCurY = CurX, CurY
\end{array}$$

The same argument advanced in Section 2.1.1 may be used here to give:

Lemma 4 : 4.1.1a

$$Rel_{Doc4}$$

$$\vdash$$

$$\forall\ ConcDoc4 \bullet \exists_1\ Doc4 \bullet Rel_{Doc4}$$

■

We calculate $AbsRel_{Doc4}$ and get, after simplification:

$$AbsRel_{Doc4} \quad \widehat{=} \quad AbsRel_{Doc3}$$

Thus we may discharge our data refinement proof obligations in exactly the same way as in Section 3.1.1, to give:

Lemma 4 : 4.1.1b

$$\vdash$$

$$Doc4 \sqsubseteq_{\infty} ConcDoc4$$

■

Finally, we calculate $ConcRel_{Doc4\,Standard}$ to give:

$$\begin{array}{|l}
\multicolumn{1}{l}{ConcRel_{Doc4\,Standard}} \\
\Delta ConcDoc4 \\
\hline
ConcRel_{Doc3\,Standard} \\
WSIns', NLIns' = WSIns, NLIns
\end{array}$$

and may again appeal (Section 3.1.1) to $[\sqsubseteq 2 : 3.2a]$ to show that $Standardize_{Doc1}$ refines this operation.

4.2 Initialisation

Specification :

$$Initialize_{Doc3} \wedge \Delta Doc4$$

Codefor $Initialize_{Doc4}$

$$Initialize_{Doc3}\ ;\ WSIns := 0\ ;\ NLIns := 0$$

Note

No new components are introduced by *ConcDoc4*; we set *WSIns* and *NLIns* to zero (although, as with *WSRem* and *NLRem* they could have any initial values).

4.3 Promotion Of The Doc3 Operations

Each abstract *OP* defined on the *Doc3* state is promoted to the *Doc4* state in the same way:

$$OP_{Doc4} \quad \widehat{=} \quad (OP_{Doc3}\ ;\ EquateQPToDoc) \wedge \Delta Doc4$$

where

EquateQPToDoc4

$$QP'$$
$$DocCurX, DocCurY : \mathbb{N}$$

$$QPCurX', QPCurY' = DocCurX, DocCurY$$

which has weakest concrete operation:

$$[\ CurX', CurY' = CurX, CurY$$

and thus represents an identity operation. The code for the *Doc3* operation will therefore provide a refinement for *Doc4*, but we must ensure that the two *ConcDoc4* components are correctly set.

4.3.1 An Operation To Update The Doc4 Components

Since each *ConcDoc3* operation cannot result in the need for the padding of whitespace/null lines, both *ConcDoc4* components must be set to zero. We thus have:

$Update_{Doc4}$

$$WSIns, NLIns \ := \ 0, 0$$

to give, as the concrete promotion mechanism for each operation OP refined on the $ConcDoc3$ state:

OP_{Doc4}

$$rep : report$$
$$rep \ := \ OP_{Doc3} \ ; \ Update_{Doc4} \ ; \ return(rep)$$

4.4 Promotion Of The QP Operations

The abstract promotion method gives:

$$CursorLeftChar_{Doc4} \ \widehat{=} \ LeftMoveChar_{Doc4}$$

and using the set $QPCursorOps$ of Section 3:5.1.1, we have

$$\begin{array}{l} \forall OP : QPCursorOps - \{CursorLeftChar\} \bullet \\ \quad OP_{Doc4} \ \widehat{=} \ FlagPrevCursor \ ; \\ \qquad\qquad SuccOP_{QP} \ ; \ PadWSNL \ ; \\ \qquad\qquad RemTrailWS \ ; \ RemTrailNL \ \land \ \Delta Doc4 \\ \qquad \lor \\ \qquad\qquad UnSuccOP_{QP} \ \land \ \Xi Doc4 \\ \qquad \lor \\ \qquad\qquad ErrorFull \ \land \ \Xi Doc4 \end{array}$$

where

$$SuccOP_{QP} \ \widehat{=} \ OP_{QP} \ | \ rep! = \text{"OK"}$$
$$UnSuccOP_{QP} \ \widehat{=} \ OP_{QP} \ | \ rep! \neq \text{"OK"}$$

$$
\begin{array}{|l}
\hline
\textit{PadWSNL} \\
\hline
\Delta \textit{UnboundedDisplay} \\
\textit{EquateDocToQP} \\
\Xi QP \\
\hline
\quad QPCurY \leq \# DocLines \\
\quad \mathsf{rng}\ (CurLine' - DocLines\ QPCurY) \subseteq \{sp\} \\
\quad \{QPCurY\} \ntriangleleft DocLines' = \{QPCurY\} \ntriangleleft DocLines \\
\lor \\
\quad QPCurY > \# DocLines \\
\quad \# DocLines' = QPCurY \\
\quad DocLines\ \mathsf{prefix}\ DocLines' \\
\quad \forall\ i : \# DocLines + 1 \mathrel{..} QPCurY - 1 \bullet DocLines'\ i = \langle\rangle \\
\quad \mathsf{rng}\ (\mathsf{last}\ DocLines') \subseteq \{sp\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{EquateDocToQP} \\
\hline
QP \\
DocCurX', DocCurY' : \mathbb{N} \\
\hline
DocCurX', DocCurY' = QPCurX, QPCurY \\
\hline
\end{array}
$$

As before, the latter is equivalent to an identity operation.

We refine the *CursorRightChar*; the refinement of the remaining operations proceed along similar lines; in so doing, we utilise operations refined on the *Doc3* state, and so post-sequential composition with *RemTrailWS* and then *RemTrailNL* will be unnecessary, since the relevant invariants will have been maintained by those *Doc3* operations. Further we use a cumulative count of whitespace and null lines removed to ensure that *WSRem* and *NLRem* reflect the total amount of whitespace/number of null lines removed by the *Doc3* operations.

Promotion Of "CursorRightChar"

Specification:

$$
\begin{array}{l}
\quad FlagPrevCursor \mathbin{;} CursorRightChar_{QP} \mathbin{;} PadWSNL \mathbin{;} \\
\quad RemTrailWS \mathbin{;} RemTrailNL \land \Delta Doc4 \land Success \\
\lor \\
\quad FlagPrevCursor \mathbin{;} ErrorFull \land \Xi Doc4
\end{array}
$$

Expansion of specification of ($CursorRightChar_{QP}$; *PadWSNL*):

$$
\begin{array}{|l}
\Delta QP \\
\Delta UnboundedDisplay \\
\hline
QPCurX', QPCurY' = QPCurX + 1, QPCurY \\
DocCurX', DocCurY' = QPCurX', QPCurY' \\
rng(CurLine' - CurLine) \subseteq \{sp\} \\
\{QPCurY'\} \lhd DocLines' = \{QPCurY'\} \lhd DocLines
\end{array}
$$

Weakest concrete operation:

$$
\left[\begin{array}{l}
CurX', CurY' = CurX + 1, CurY \\
CurX', CurY' = CurX + 1, CurY \\
\mathsf{rng}\ ((StartIn' + 1 \mathinner{..} EndIn' \upharpoonleft ArrCont') - \\
\qquad\qquad (StartIn + 1 \mathinner{..} EndIn \upharpoonleft ArrCont)) \subseteq \{sp\} \\
\{CurY'\} \lhd (\mathsf{FL}^{-1}\ ArrCont') = \{CurY\} \lhd (\mathsf{FL}^{-1}\ ArrCont)
\end{array}\right.
$$

Simplification and refinement:

$$
\left[\begin{array}{l}
RP', StartIn', CurY', NLIns' = RP, StartIn, CurY, 0 \\
CurX' = CurX + 1 \\
CP' = CP + 1 \\
LP - RP = LP' - RP' + WSRem' + NLRem' - WSIns' - NLIns' \\
\quad CP \neq EndIn \\
\quad LP', EndIn', Arr' = LP, EndIn, Arr \\
\quad WSIns' = 0 \\
\vee \\
\quad CP = EndIn \wedge LP \neq RP \\
\quad CP' = LP' = CP + 1 \\
\quad EndIn' = EndIn + 1 \\
\quad Arr' = Arr \oplus \{CP' \mapsto sp\} \\
\quad WSIns' = 1
\end{array}\right.
$$

Simplification and refinement of weakest concrete operation for $\Xi Doc4$:

$$
\left[\begin{array}{l}
NoChange(ConcDoc4) \\
CP = EndIn \wedge LP = RP \\
CP', ArrCont' = CP, ArrCont \\
WSIns', NLIns' = 0, 0
\end{array}\right.
$$

Code for $CursorRightChar_{Doc4}$

$rep : Report$;
$WSIns := 0$; $NLIns := 0$;
if

$$
\begin{array}{lll}
(CP \neq Endln) & \longrightarrow & OP := RightMoveChar \,;\, rep := OP_{Doc3} \,; \\
 & & \{\, WSRem = NLRem = WSIns = NLIns = 0 \,\} \\
 & & \{\, LP_o - RP_o = LP - RP \,\} \\
\Box & & \\
(CP = Endln) & \longrightarrow & OP := RightMoveChar \,;\, OPChar := sp \,; \\
 & & rep := OP_{Doc3} \,;\, WSIns{+}{+} \,; \\
 & & \{\, WSRem = NLRem = NLIns = 0 \,\} \\
 & & \{\, LP_o - RP_o = LP - RP - WSRem \,\}
\end{array}
$$

fi
$OP := CursorRightChar$; $return(rep)$
$\{\, LP_o - RP_o = LP - RP + WSRem + NLRem - WSIns - NLIns \,\}$

Notes

Specification:
We note that the *QP* operation is always successful.

Expansion of specification of $(CursorRightChar_{QP}\ ;\ PadWSNL)$:
Since the cursor line does not change, $(DocLines\ QPCurY')$ will be the same as *CurLine*.

Simplification and refinement:
We choose to introduce two pre-conditions, the first corresponding to the cursor not being at the end of the line, and the second corresponding to the cursor being at the end of the line and with capacity to insert a space character. thus splitting the first part of the specification into two disjuncts. A third pre-condition (ensuring a total operation), corresponding with the cursor being at the end of the current line but with the editor being full, is introduced for the final part of the refinement. We appeal to [$\sqsubseteq$ 2 : 3.5.2b] using each pre-condition as the guard for the alternate construct, noting that each body is total.

When the cursor resides at the end of the current line, (*CP* is equal to *Endln*), since the cursor line and hence *Startln* do not change, we may refine the fourth predicate of the weakest concrete operation to:

$$Startln + 1 \,..\, Endln \lhd ArrCont' = Startln + 1 \,..\, Endln \lhd ArrCont$$
$$ArrCont'\ CP' = sp$$

which, together with the final predicate and assuming a standardized array (ensured by *InsertChar*) it is refined by:

$$Arr' = Arr \oplus \{CP' \mapsto sp\}$$

Note that in this case it is necessary to increment the value of *Endln* in order to maintain the *ConcDoc3* invariant.

Conversely, when the cursor is not at the end of the line, that same invariant requires that *Endln* does not change, and the same predicates of the weakest concrete operation are refined by leaving the entire array unchanged.

Code:

The requirements of the first disjunct are satisfied by the *Doc3* right character move; those for the second and "no change" disjuncts are satisfied by the *Doc3* insertion of a space character, that operation providing the check for a full array. Further, since the *Doc3* operations do not necessitate the removal of trailing whitespace/null lines, both *WSRem* and *NLRem* are zero after the operation.

5 Refinement Of Doc6

The *Doc5* state introduces marked text and the *SetMark* operation: however we choose to also include the paste buffer and its operations in our next refinement step, since we feel that the consideration of marked text alone does not merit a refinement level. Thus we now consider refinement on the *Doc6* state.

5.1 The Design Decision

The *Doc4* state is enlarged to the *Doc6* state by the inclusion of the abstract components *PBuff*, *MarkSeq* and *MarkedSeq*.

Our concrete representation for the first is the character array *PArr* of size *Max* and pointer *PP*, a natural number not exceeding *Max*. We introduce the pointer *MP*, lying between -1 and *Max* with -1 indicates that no text is marked. Otherwise, if *MP* is less than *CP MarkSeq* and *MarkedSeq* are equal to the *ArrCont* up to *MP* and the *ArrCont* lying between *MP* and *CP* respectively; if *MP* exceeds *CP* they are equal to the *ArrCont* after *MP* and the *ArrCont* lying between *CP* and *MP* respectively:

ConcPasteBuffer

$$PArr : 1 \,..\, Max \rightarrow Char$$
$$PP : 0 \,..\, Max$$

$$ConcMark \;\hat{=}\; MP : -1 \,..\, Max$$

$$ConcDoc6 \;\hat{=}\; ConcDoc4 \wedge ConcPasteBuffer \wedge ConcMark$$

5.1.1 The Concrete-Abstract Invariant

Rel_{Doc6}

$Doc6$
$ConcDoc6$
Rel_{Doc4}

$MP = -1 \;\Rightarrow\; MarkSeq = MarkedSeq = \langle\rangle$
$MP \neq -1 \;\wedge\; MP \leq CP \;\Rightarrow\; MarkSeq = ArrCont \text{ for } MP$
$\qquad MarkedSeq = MP + 1 \mathinner{..} CP \upharpoonleft ArrCont$
$MP \neq -1 \;\wedge\; MP > CP \;\Rightarrow\; MarkSeq = ArrCont \text{ after } MP$
$\qquad MarkedSeq = CP + 1 \mathinner{..} MP \upharpoonleft ArrCont$
$PBuff = PArr \text{ for } PP$

In order to satisfy the condition that every concrete state corresponds to a unique abstract state [$\sqsubseteq$ 2 : 1.1.1*a*], we have only to show that the concrete paste buffer defines a unique abstract one, and similarly for the concrete mark (the *ConcDoc4* obligation was discharged in Section 4.1.1): clearly both follow immediately from Rel_{Doc6}, to give:

Lemma 4 : 5.1.1a

Rel_{Doc6}

$\vdash$

$\forall\; ConcDoc6 \bullet \exists_1\; Doc6 \bullet Rel_{Doc6}$

■

We calculate *AbsRel*, and have, after simplification:

$$AbsRel_{Doc6} \;\widehat{=}\; [\; \Xi Doc6 \;\wedge\; AbsRel_{Doc4} \mid \#\, PBuff \leq Max \;]$$

Using the same argument as that in Section 2.1.1, since:

$$\lim_{Max \to \infty} (\#\,(PBuff \leq Max) \;\equiv\; \mathsf{true})$$

we discharge our data refinement proof obligation by appealing to [$\sqsubseteq$ 2 : 4.1*b*] and the above lemma, implying an ∞-refinement:

Lemma 4 : 5.1.1b

$\vdash$

$Doc6 \;\sqsubseteq_{\infty}\; ConcDoc6$

■

We now calculate $ConcRel_{Doc6\,Standard}$ to give:

$$
\begin{array}{|l}
\hline
ConcRel_{Doc6\,Standard} \\
\Delta ConcDoc6 \\
\hline
ConcRel_{Doc6\,Standard} \\
PArr' \text{ for } PP' = PArr \text{ for } PP \\
MP' = MP \\
\hline
\end{array}
$$

and may again appeal (Section3.1.1) to $[\sqsubseteq 2 : 3.2a]$ to show that $Standardize_{Doc1}$ refines this operation.

5.2 Refinement Of "SetMark"

Specification :

$$setMk \ \wedge \ \Xi Doc4 \ \wedge \ \Delta Doc5 \ \wedge \ \Xi PasteBuffer \ \wedge \ Success$$

where:

$$
\begin{array}{|l}
\hline
SetMk \\
MarkedText' \\
Pair_{Char} \\
\hline
MarkSeq' = Left_{Char}{}' \\
MarkedSeq' = \langle\,\rangle \\
\hline
\end{array}
$$

Simplification and refinement of weakest concrete operation:

$$
\left[
\begin{array}{l}
NoChange(ConcDoc6 \setminus ConcMark) \\
MP' = CP \\
rep! = \text{"OK"}
\end{array}
\right.
$$

$Mark_{Doc6}$

$$MP \ := \ CP \ ; \ return(\text{"OK"})$$

5.3 Refinement Of "Lift"

Specification

$$CopyMTextPBuff \ \wedge \ \Xi Doc5 \ \wedge \ \Delta Doc6 \ \wedge \ Success$$

$\vee$

$$ErrorNoTextMarked$$

where:

CopyMTextPBuff

$\Delta PasteBuffer$
$Left_{Char}$
$MarkedText$

$PBuff' = MarkedSeq \neq <>$

Simplification of weakest concrete operation for *CopyMTextPBuff*:

$$\left[\begin{array}{lll} CP \neq MP \neq -1 & & \\ CP < MP & \Rightarrow & PArr' \text{ for } PP' = CP + 1 \,..\, MP \upharpoonright ArrCont \\ CP > MP & \Rightarrow & PArr' \text{ for } PP' = MP + 1 \,..\, CP \upharpoonright ArrCont \end{array}\right.$$

CopyMTextPBuff

$MPptr : 0 \,..\, Max$;
$\{ CP \neq MP \wedge MP \neq -1 \}$
$PP := 0$;
if
| $(CP < MP) \longrightarrow CopyMTextPBuff\_CurAbove$
$\Box$
| $(CP > MP) \longrightarrow CopyMTextPBuff\_CurBelow$
fi

CopyMTextPBuff_CurAbove

$MPptr := CP$;
do
| $(MPptr \neq MP) \longrightarrow MPptr{+}{+}$; $PP{+}{+}$; $PArr\ PP := GetArrCont(MPptr)$
| $\{$ Invariant : $PArr$ for $PP = MPptr + 1 \,..\, CP \upharpoonright ArrCont \}$
| $\{$ Variant : $MP - MPptr \}$
| $\{$ Guard Negation : $MPptr = MP \}$
od

Code for $Lift_{Doc6}$

$FlagCPLPRP$;
$rep : Report$;
if
| $(CP \neq MP \ \wedge \ MP \neq -1) \longrightarrow$ $CopyMTextPBuff$; $return$("OK")
$\Box$
| $(CP = MP \ \vee \ MP = -1) \longrightarrow$ $return$("No text marked")
fi ;
$Update_{Doc3}(prevCP, prevLP, prevRP)$; $Update_{Doc4}$; $return(rep)$

Note

Code

We appeal to [$\sqsubseteq$ 2.3.5.2*b*], the pre-condition for each disjunct providing the guard; the loop is, of course, total, and for which we use [$\sqsubseteq$ 2.3.5.3*a*]. The code for the body of the second disjunct, *CopyMTextPBuff CurBelow*, is similar to that for *CopyMTextPBuff CurAbove*.

5.4 Refinement of "Paste"

Specification

$FlagPrevCursor$;
  Pst ; $RemTrailWS$; $RemTrailNL \ \wedge \ \Delta Doc6 \ \wedge \ Success$
$\vee$
  $\Xi Doc6 \ \wedge \ ErrorFull$
$\vee$
  $ErrorPBuffEmpty$

where:

Pst
$\Delta Pair_{Char}$
$\Delta MarkedText$
$PasteBuffer$

$PBuff \neq \langle\ \rangle$
$Left_{Char}' = Left_{Char} \frown PBuff$
$Right_{Char}' = Right_{Char}$
$MarkSeq' = Left_{Char}$
$MarkedSeq' = \langle\ \rangle$

We first consider *Pst* and have, as the simplification and refinement of its weakest *Standard* concrete operation:

$$\left[\begin{array}{l} PP \neq 0 \\ PP \leq RP - LP \\ LP, LP' = CP, CP' \\ Arr' \text{ for } LP' = Arr \text{ for } LP \frown PArr \text{ for } PP \\ Arr' \text{ after } LP' = Arr \text{ after } LP \\ MP' = LP \end{array}\right.$$

Pst

$\{ LP = CP \wedge PP \neq 0 \wedge PP \leq RP - LP \}$
if
| $MP := CP$; *Pst1*
fi

Pst1

$PPptr := 0$;
do
| $(PPptr \neq PP) \longrightarrow PPptr{+}{+}$; $LP{+}{+}$; $CP{+}{+}$; $Arr\ LP := PArr\ PPptr$
  $\{$ Invariant : Arr for $LP = Arr_o$ for $LP_o \frown PArr_o$ for $PPptr \}$
  $\{$ Variant : $PP - PPptr \}$
  $\{$ Guard Negation : $PPptr = PP \}$
od

Code for $Paste_{Doc6}$

FlagCPLPRP ;
$PPptr : 0 \,..\, Max$; $rep : Report$;
if
| $(PP = 0) \longrightarrow rep$:= "Paste buffer empty"
□
| $(PP \neq 0 \wedge PP > RP - LP) \longrightarrow rep$:= "Editor full"
□
| $(PP \neq 0 \wedge PP \leq RP - LP) \longrightarrow Standardize_{Doc1}$; *Pst* ; rep := "OK"
fi ;
$Update_{Doc3}(prevCP, prevLP, prevRP)$; $Update_{Doc4}$; $return(rep)$

Note

Weakest concrete operation:

We appeal to [$\sqsubseteq$ 2 : 2.2*b*] to refine on a *Standard* configuration, (pushing the *Standardize* operation through to where it is actually required).

Code:

The pre-condition that PP must not exceed the difference between RP and LP for the third conjunct is implied by the *ConcDoc1* invariant, and we appeal to $[\sqsubseteq 2 : 3.3a]$ (treating the first two disjuncts as a single operation) to refine to a disjunct noting that one pre-condition is the negation of the other. We then use $[\sqsubseteq 2 : 3.5.2b]$, with each pre-condition providing the guard, and appeal to $[\sqsubseteq 2 : 2.5.3a]$ for the refinement of *Pst1* to a loop.

5.5 Promotion Of The Doc4 Operations

The abstract promotion method for the *Doc4* operations to the *Doc5* state requires that for content-changing operations the mark be re-set, but is non-deterministic for cursor-changing operations; the subsequent promotion to *Doc6* requires a no-change paste buffer. Using the sets *CursorOps_NoMarkSet*, *CursorOps_MarkSet* and *NonCursorOps* (Part 3, Section 5.1.1), we have:

$$\forall\ OP : NonCursorOps\ \bullet$$
$$OP_{Doc6}\ \widehat{=}\ OP_{Doc4}\ \wedge\ ResetMark\ \wedge\ \Xi PasteBuffer$$
$$\forall\ OP : CursorOps\_NoMarkSet\ \bullet$$
$$OP_{Doc6}\ \widehat{=}\ OP_{Doc4}\ \wedge\ \Xi MarkedText\ \wedge\ \Xi PasteBuffer$$
$$\forall\ OP : CursorOps\_MarkSet\ \bullet$$
$$OP_{Doc6}\ \widehat{=}\ OP_{Doc4}\ \wedge\ \Delta MarkedText\ \wedge\ \Xi PasteBuffer$$

Since it is desirable that cursor-changing operations should preserve the mark position, we choose to do that provided, after the operation, the mark does not point beyond the bottom of the document (as it may do after the removal of trailing null lines); if it does, the mark is re-set.

We therefore have:

Code for $Update_{Doc6}(prevCP, prevLP, prevRP)$ $\quad$ $prevCP?, prevLP?, prevRP? : 0 \,..\, Max$

if

$((OPType \neq NoMove\ \wedge\ OPType \neq LeftMove\ \wedge\ OPType \neq RightMove) \vee$
$\quad MP > Max + LP - RP) \longrightarrow MP := -1$

$[]$

$((OPType = NoMove\ \vee\ OPType = LeftMove\ \vee\ OPType = RightMove) \wedge$
$\quad MP \leq Max + LP - RP\ \wedge\ MP \neq -1) \longrightarrow AdjustMark$

$[]$

$((OPType = NoMove\ \vee\ OPType = LeftMove\ \vee\ OPType = RightMove) \wedge$
$\quad MP \leq Max + LP - R\ \wedge\ MP = -1) \longrightarrow Skip$

fi

AdjustMark

if

$(prevCP \leq MP \wedge CP \leq MP) \rightarrow MP := MP - WSRem + WSIns$

$\Box$

$(prevCP \leq MP \wedge CP \geq MP) \rightarrow MP := MP - WSRem$

$\Box$

$(prevCP \geq MP \wedge CP \leq MP) \rightarrow MP := MP + WSIns$

$\Box$

$(prevCP \geq MP \wedge CP \geq MP) \rightarrow Skip$

fi

We now have the concrete promotion method for all operations *OP* defined on the *ConcDoc4* state:

OP_{Doc6}

$FlagCPLPRP$;
$rep := OP_{Doc4} : Report$;
$Update_{Doc6}(prevCP)$; $return(rep)$

Note
If the cursor is originally above the marked position, the latter's relative position will change by the amount of whitespace removed (and thus we must reduce *MP* by *WSRem*), and is the final position of the cursor is above the marked position, the latter's relative position will change by the amount of whitespace inserted (and so we need to increase *MP* by the *WSIns*). The insertion of newlines (by *PadWSNL* can only take place when the cursor is moved below the bottom of the document (and so *MP* will be less than *CP*, which requires no change in the former to maintain its relative position), and the only way in which the removal of newlines (by *RemTrailNL*) can affect *MP* is if it points to one of those newlines, and so will end up pointing to a point beyond the bottom of the document (and thus will be reset).

We use [$\sqsubseteq$ 2 : 3.5.2*b*] for the *Update* alternate construct, and [$\sqsubseteq$ 2 : 3.3*a*] for that of *AdjustMark*, noting that the disjunct of the pre-conditions is *true*.

6 Refinement Of Doc8

The *Doc7* state introduces the quote buffer, together with an editor state (to enable switching between document and quote buffer editing), and a document file name, with the quote commands specified on this state. The *Doc8* state introduces search and replace buffers, enabling the specification of the search and replace operations. Since the latter two operations may also be regarded as quote operations (they may be effected from either editor state), we choose not to refine on the *Doc7* state, but to incorporate all of the quote operations in one single refinement step, on the *Doc8* state.

6.1 The Design Decision

We represent the left and right quote buffers by a single character array, *QArr*, together with the pointers *QP* (*Q*uote *P*ointer) and *QCP* (*Q*uote *C*ursor *P*ointer). *QBuff*, the concatenation of the left and right quote buffers, is redundant, and we choose to exclude it from the concrete state.

We identify *QMax* as a *ResourceLimit*, although, clearly, the specifier intended the implementation to provide only a small buffer.

It is envisaged that the quote buffer will displayed on a single line of the terminal screen, and its signature (together with those of the search and replace buffers) will reflect this. Although we use an array and pointers as we did with the *Doc1* representation, the quote buffer content will be represented by the first *QP* locations of the array (a zero pointer meaning an empty array), with *QCP* representing the cursor position; the small size of the array will render array shuffling inexpensive.

The search and replace buffers will be represented by the character arrays *SArr* and *RpArr*, together with their pointers *SP* and *RpP* respectively, having minimum value *0* and maximum value *QMax* - also *ResourceLimits* - with *MatchedLength* equal to the number of characters following the cursor that the search buffer corresponds to when the document is in a matched state (since, because of the use of regular expressions, we cannot take the length of the buffer). The document filename is represented by the string *FName*, the concrete editor state being *EState*, sharing the signature of its abstract counterpart.

We also include the boolean *DocChanged*, indicating whether or not the document's content has changed since the last read from/write to store.

QMax : *ResourceLimit*

ConcDoc8

$ConcDoc6$
$QArr, SArr, RpArr : 1 \,..\, QMax \rightarrow Char$
$QP, QCP, SP, RpP : 0 \,..\, QMax$
$FName : String$
$MatchedLength : 0 \,..\, QMax$
$EState : \{ State_{Doc}, State_{Quote} \}$
$DocChanged : \mathbf{B}$

$QCP \leq QP$
$(SArr \text{ for } SP) \text{ matches } (ArrCont \text{ after } CP) \Rightarrow$
$\quad MatchedLength = \text{matchedlength } (SArr \text{ for } SP, ArrCont \text{ after } CP)$

6.1.1 The Concrete-Abstract Invariant

Rel_{Doc8}

$Doc8$
$ConcDoc8$
Rel_{Doc6}

$State = EState$
$Left_{Quote} = QArr \text{ for } QCP$
$Right_{Quote} = QCP + 1 \,..\, QP \upharpoonright QArr$
$SBuff = SArr \text{ for } SP$
$RBuff = RpArr \text{ for } RpP$
$QBuff = QArr \text{ for } QP$
$FileName = FName$

Hence, for a given concrete state Rel_{Doc8} will uniquely define the abstract *Doc8* components,a to give:

Lemma 4 : 6.1.1a

Rel_{Doc8}
$\vdash$
$\forall\ ConcDoc8 \bullet \exists_1\ Doc8 \bullet Rel_{Doc8}$

■

We calculate $AbsRel$, and after simplification obtain:

$AbsRel_{Doc8}$

$\Xi Doc8$
$AbsRel_{Doc6}$

$SBuff \leq QMax$
$RBuff \leq QMax$
$QBuff \leq QMax$

We proceed exactly as in Section 5.1.1 to obtain:

Lemma 4 : 6.1.1b

$\vdash$

$Doc8 \ \sqsubseteq_{\infty} \ ConcDoc8$

■

$AbsRel_{Doc8} \ \hat{=} \ \Xi Doc8 \ \wedge \ AbsRel_{Doc7}$

$ConcRel_{Doc8}$

$\Delta ConcDoc8$
$ConcRel_{Doc6}$

$QArr'$ for $QP' = QArr$ for QP
$SArr'$ for $SP' = SArr$ for SP
$RpArr'$ for $RpP' = RpArr$ for RpP
$FName' = FName$
$QCP', EState', DocChanged' = QCP, EState, DocChanged$

Similar comments to those made in Section 6.1.1 also apply here.

6.2 Refinement Of I/O Operations

In the specification we make broad assumptions concerning the operations to return a pointer to a file, *GetSysPtr*, and those to read and write a file, *SysRead* and *SysWrite* respectively, and specify the operations *ReadFromStore* and *WriteToStore* (making use of those i/o operations) which incorporate the return of appropriate reports indicating the success, or otherwise, of the i/o operations.

Refinement Of "ReadFromStore"

Specification :

$$SuccSysGetPtr \gg SuccSysRead \wedge Success$$
$$\vee$$
$$SuccGetSysPtr \gg UnSuccReadFile \wedge ErrorReadingFile$$
$$\vee$$
$$UnSuccSysGetPtr \wedge ErrorCannotOpenFile$$
$$\vee$$
$$SuccSysGetPtr \wedge ErrorFull$$

SysRead

$\Delta Store$
$SysPtr? : \mathbf{N}$
$ReadSeq! : seq\ Char$
$NoReadError! : \mathbf{B}$

Code for *ReadFromStore*(*filename*) *filename*? : *String*

```
filelength : N ;
FlagPrevDoc1 ;
SysPtr := fopen (filename, < r >) : N ;
if
| (SysPtr ≠ NullPtr) →
|         if
|         | (FileSuitable(filename)) →
|         |         filelength := SysGetFilelength(filename) ;
|         |         if
|         |         | (filelength ≤ RP − LP) → ReadToCurrentPosition
|         |         []
|         |         | (filelength > RP − LP) → return("Editor full")
|         |         fi
|         []
|         | (¬ FileSuitable(filename)) → return("Unsuitable file")
|         fi
[]
| (SysPtr = NullPtr) → return("Cannot open file")
fi
```

ReadToCurrentPosition

$NoReadError$:= **true** : $\mathbb{B}$;

$Standardize_{Doc1}$;

do

| $(LP < RP \ \wedge \ NoReadError \ \wedge \ NoInterrupt \ \wedge \ filelength > 0) \longrightarrow$

$GetNextNonCntrlChar$; $Arr(LP + 1)$:= x ; $LP{+}{+}$;

if

| $(x = nl) \longrightarrow StripTrailWS$

$\square$

| $(x \neq nl) \longrightarrow Skip$

fi ;

{ Invariant : Read($LP_o + 1 \, .. \, LP \upharpoonleft Arr$) }

{ Invariant : $\forall \ l$: $FL^{-1}(CP \, .. \, LP \upharpoonleft Arr) \ \bullet$ visible l }

{ Variant : $filelength$ }

{ Guard Negation : $LP \geq RP \ \vee \ \neg NoReadError \ \vee \ \neg NoInterrupt \ \vee \ filelength = 0$ }

od ;

fclose ($SysPtr$) ;

if

| $(LP = RP) \longrightarrow LP := prevLP$; *return*("Editor full")

$\square$

| $(\neg NoReadError) \longrightarrow LP := prevLP$; *return*("Error reading file")

$\square$

| $(filelength = 0) \longrightarrow$ { $ReadSeq = LP_o + 1 \, .. \, LP \upharpoonleft Arr$ }

$CP := LP$; *return*("OK")

fi

GetNextNonCntrlChar

$cntrlfnd$: $\mathbb{B}$;

x := getc ($SysPtr$) ; $filelength{-}{-}$; $cntrlfnd := ControlChar \ x$;

do

| $(cntrlfnd \ \wedge \ filelength > 0 \ \wedge \ NoInterrupt) \longrightarrow$

x := getc ($SysPtr$) ; $filelength{-}{-}$; $cntrlfnd := ControlChar \ x$

{ Invariant : $\forall \ c : Store \ (\! | \ SysPtr_o \, .. \, SysPtr - 1 \ | \!) \ \bullet \ ControlChar \ c$ }

{ Variant : $filelength$ }

{ Guard Negation : $\neg cntrlfnd \ \vee \ filelength = 0 \ \vee \ \neg NoInterrupt$ }

od

{ $filelength > 0 \ \wedge \ NoInterrupt \ \Rightarrow \ \neg (ControlChar \ x)$ }

StripTrailWS

do
$(LP > CP + 1 \wedge Arr(LP - 1) = sp) \longrightarrow LP{-}{-}\,;\, Arr\ LP := nl$
{ Invariant : $Arr_o \llparenthesis LP + 1 \,..\, LP_o - 1 \rrparenthesis \subseteq \{sp\}$ }
{ Invariant : $Arr\ LP = nl$ }
{ Variant : $LP - (CP - 1)$ }
{ Guard Negation : $LP = CP + 1 \vee Arr(LP - 1) \neq sp$ }
od
{ last (FL^{-1} $(CP \,..\, LP \upharpoonleft Arr)) = \langle\rangle$ }
{ $LP \neq CP + 1 \Rightarrow$ visible (last FL^{-1} $(CP \,..\, LP - 1 \upharpoonleft Arr)$) }

Note
For the first disjunct of *ReadFromStore*, we introduce the pre-condition that there must be sufficient editor capacity to accommodate the file (i.e. the file length must not exceed the difference between *RP* and *LP*); in order to preserve the totality of the operation, we use the negation of this pre-condition as that for the last disjunct, and appeal to [$\sqsubseteq$ 2 : 3.3*a*] for the alternate construct.

Since a successful *ReadFromStore* is always piped into *InputReadSeq* (see Section 6.4.1), and the latter requires that the sequence of characters read is inserted into the document at the current position, we refine the former so that that requirement is satisfied. *StripTrailWS* ensures that *ReadSeq* satisfies the whitespace invariant of *Doc3*, and the final *Update* operations (Section 6.5) ensure that the original cursor line also satisfies that invariant, and that trailing null lines are removed.

We use the **C** function putc to return a character from a file, and the informal boolean **Read** to indicate which part of *ReadSeq* has been read from store and inserted into the array. The function *SysGetFilelength* is assumed to return the length of the file (using the **C** call **stat**) and we assume that *ControlChar* indicates whether or not a character is a valid text character or not. In the implementation we expand the *TAB* control character to an appropriate number of spaces (with a test to ensure that the editor capacity is not exceeded); the procedure follows that for the insertion of a *tab* character in the refinement of *InsertChar*, Section 2.3.4.

We recognize that the user may interrupt the reading of a file, when, we assume that the *NoInterrupt* flag is set to **false**, and in this case a read error message is reported.

6.3 Refinement Of The Quote Operations

We refine the operations *Input* and *DownSearch* to indicate the method of refinement for operations specified on the *Doc7* and *Doc8* states.

6.3.1 Refinement Of "Input"

Specification of $Input_{Doc7}$:

$$
\begin{array}{l}
FlagPrevCursor\ ; \\
\quad (RequestInput \gg SuccReadFromStore \gg InputReadSeq)\ ; \\
\quad RemTrailWS\ ;\ RemTrailNL \\
\vee \\
\quad RequestInput \gg UnSuccReadFromStore \\
\vee \\
\quad RequestInput \wedge \Xi Doc5 \wedge ErrorFileNotExist \\
\vee \\
\quad RequestInput \wedge \Xi Doc5 \wedge UnsuitableFile \\
\vee \\
\quad RequestInput \wedge \Xi Doc5 \wedge ErrorFull
\end{array}
$$

$$Input_{Doc8} \mathrel{\widehat{=}} Input_{Doc7} \wedge \Xi SearchBuffer \wedge \Xi ReplaceBuffer$$

Expansion of $(RequestInput \gg SuccReadFromStore \gg InputReadSeq)$:

$$
\begin{array}{|l}
\Delta Doc7 \\
\Xi DocName \\
\Xi PasteBuffer \\
\Xi QuoteBuffer \\
ReadSeq? : \mathrm{seq}\ Char \\
filename! : String \\
filemode! : \{<r>, <w>, <a>\} \\
\hline
State = State_{Quote} \\
State' = State_{Doc} \\
<i, sp>\ \mathsf{prefix}\ QBuff \\
filename! = QBuff\ \mathsf{after}\ 2 \\
filemode! = <r> \\
Mark' = -1 \\
Left_{Char}' = Left_{Char} \frown ReadSeq? \\
Right_{Char}' = Right_{Char}
\end{array}
$$

Weakest concrete operation:

$$\left[\begin{array}{l} State\_ConcDoc8 \\ NoChange(ConcDoc8 \setminus ConcDoc2, MP, EState) \\ EState = State_{Quote} \\ EState' = State_{Doc} \\ < i, sp > \ \mathsf{prefix}_1 \ (QArr \ \mathsf{for} \ QP) \\ MP' = -1 \\ ArrCont' \ \mathsf{for} \ CP' = ArrCont \ \mathsf{for} \ CP \frown ReadSeq? \\ ArrCont' \ \mathsf{after} \ CP' = ArrCont \ \mathsf{after} \ CP \end{array}\right.$$

Simplification and refinement:

$$\left[\begin{array}{l} State\_ConcDoc8 \\ NoChange(ConcDoc8 \setminus ConcDoc2, MP, EState) \\ \# ReadSeq? \leq RP - LP \\ EState = State_{Quote} \\ EState' = State_{Doc} \\ QP \geq 3 \\ QArr \ \mathsf{for} \ 2 = < i, sp > \\ MP' = -1 \\ LP', CP', RP' = LP + \# ReadSeq?, CP + \# ReadSeq?, RP \\ ArrCont' = ArrCont \ \mathsf{for} \ CP \frown ReadSeq? \frown ArrCont \ \mathsf{after} \ CP \end{array}\right.$$

Code:

$Input_{Doc8}$

$\{ EState = State_{Quote} \}$
$\{ QP \geq 3 \ \wedge \ QArr \ \mathsf{for} \ 2 = < i, sp > \}$
$rep : Report \ ; \ filename : String \ ;$
$EState := State_{Doc} \ ; \ OPType := LeftInsert \ ; \ CopyQBuffToString(filename, 2) \ ;$
$\{ filename = QBuff \ \mathsf{after} \ 2 \}$
$rep := ReadFromStore(filename) \ ;$
if
$\mid (rep = \text{“OK”}) \longrightarrow DocChanged := \mathsf{true} \ ; \ MP := -1$
$\square$
$\mid (rep \neq \text{“OK”}) \longrightarrow Skip$
fi

Note

Since *ReadFromStore* ensures the whitespace invariant (and the operation cannot violate the null lines invariant), we can dispense with the *Rem* operations of the first disjunct. We again use [$\sqsubseteq$ 2 : 3.3*a*] and [$\sqsubseteq$ 2 : 3.5.2*b*] to refine to the alternate construct.

Since we refined a succesful *ReadFromStore* to concateuate the seqneuce of characters read on to the end of the left character sequence, the last two predicates of the refinement are satisfied. The implementation-dependent *CopyQBuffToString* copies the quote buffer contents from the third (one more than the value of the second parameter) location to the QP^{th} location to the string *filename* (the value of the first parameter). In the implementation we split the "unsuitable file" disjunct into two: we check to see whether or not the file is a directory, and whether or not the file has read permission, with appropriate reports returned if either requirement is not satisfied.

6.3.2 Refinement Of "DownSearch"

Specification:

$$
\begin{array}{l}
FlagPrevCursor\ ;\\
\quad SuccDownSearch\ ;\ RemTrailWS\ ;\ RemTrailNL\ \wedge\\
\quad PromoteSearchUnMark\ \wedge\ Success\\
\vee\\
\quad UnSuccDownSearch\ \wedge\ PromoteSearchLeaveMark\ \wedge\ RepStringNotFound\\
\vee\\
\quad ErrorSBuffEmpty
\end{array}
$$

where

$$
\begin{array}{lcl}
SuccDownSearch & \widehat{=} & SuccDownSch_{Doc}\ \vee\ SuccDownSch_{Quote}\\
UnSuccDownSearch & \widehat{=} & UnSuccDownSch_{Doc}\ \vee\ UnSuccDownSch_{Quote}\\
\\
SuccDownSearch_{Quote} & \widehat{=} & CopyQBuffSBuff\ ;\ SuccDownSch_{Doc}\\
UnSuccDownSearch_{Quote} & \widehat{=} & CopyQBuffSBuff\ ;\ UnSuccDownSch_{Doc}
\end{array}
$$

Expansion of *SuccDownSearch*:

SuccDownSearch

$$
\begin{array}{l}
\Delta Doc8\\
\Xi_{Cont} Doc1\\
\hline
SBuff' = SBuff \neq \langle\rangle\\
State = State' = State_{Doc}\\
SBuff \text{ matches } Right_{Char}'\\
\neg(\exists\ S \text{ in } n\,..\,m \upharpoonright (Left_{Char} \frown Right_{Char}) \bullet SBuff \text{ matches } S)\\
\text{where}\\
\quad n, m = (\#\,Left_{Char} + 2).(\#\,Left_{Char}' + \text{matchedlength}\,(SBuff, Right_{Char}') - 1)
\end{array}
$$

Simplification and refinement of weakest concrete operation of first conjunct:

$$\left[\begin{array}{l} \mathit{StateConcDoc8} \\ \mathit{NoChangeConcDoc8} \setminus (\mathit{ConcDoc2}, \mathit{EState}, \mathit{SP}, \mathit{SArr}) \\ \mathit{SP}' = \mathit{SP} \neq 0 \\ \mathit{SArr}' = \mathit{SArr} \\ \mathit{EState} = \mathit{EState}' = \mathit{State}_{Doc} \\ (\mathit{SArr} \text{ for } \mathit{SP}) \text{ matches } (\mathit{CP}' + 1 \mathinner{..} \mathit{CP}' + \mathit{SP} \upharpoonleft \mathit{ArrCont}) \\ \neg (\mathit{SArr} \text{ for } \mathit{SP} \text{ matches } (\mathit{CP} + 2 \mathinner{..} \mathit{CP}' + \mathit{SP} - 1 \upharpoonleft \mathit{ArrCont})) \end{array}\right.$$

Expansion of *UnSuccDownSearch*:

UnSuccDownSearch

$$\Xi Doc8$$

$$\begin{array}{l} \mathit{SBuff} \neq \langle\rangle \\ \mathit{State} = \mathit{State}_{Doc} \\ \neg (\exists\ S \text{ in } (\text{tail } \mathit{Right}_{Char}) \bullet \mathit{SBuff} \text{ matches } S) \end{array}$$

Simplification and refinement of weakest concrete operation for second conjunct:

$$\left[\begin{array}{l} \mathit{StateConcDoc8} \\ \mathit{NoChangeConcDoc8} \\ \mathit{SP} \neq 0 \\ \mathit{EState} = \mathit{State}_{Doc} \\ \neg (\exists\ S \text{ in } (\mathit{ArrCont} \text{ after } \mathit{CP} + 1) \bullet (\mathit{SArr} \text{ for } \mathit{SP}) \text{ matches } S) \end{array}\right.$$

Code for $\mathit{DownSearch}_{Doc8}$

$$\begin{array}{l} \mathit{matched} := \mathsf{false} : \mathbf{B}\ ; \qquad \mathsf{prevCP} := \mathsf{CP} : 0 \mathinner{..} \mathbf{Max}\ ; \\ \mathit{OPType} := \mathit{RightMove}\ ; \\ \mathbf{if} \\ \mid (\mathit{EState} = \mathit{State}_{Quote}) \longrightarrow \mathit{CopyQBuffSBuff}\ ;\ \mathit{EState} := \mathit{State}_{Doc} \\ \Box \\ \mid (\mathit{EState} \neq \mathit{State}_{Quote}) \longrightarrow \mathit{Skip} \\ \mathbf{fi}\ ; \\ \mathbf{if} \\ \mid (\mathit{SP} \neq 0) \longrightarrow \mathit{DownSearch}\ 1 \\ \Box \\ \mid (\mathit{SP} = 0) \longrightarrow \mathit{rep} := \text{“Search buffer empty”} \\ \mathbf{fi} \end{array}$$

do

| $(CP < Max + LP - RP \wedge \neg matched \wedge NoInterrupt) \longrightarrow$

| $\quad CP{+}{+}\,;\ CheckForMatch$

| { Invariant : $matched \Rightarrow (SArr$ for $SP)$ matches $(ArrCont$ after $CP)$ }

| { Invariant : $\neg matched \Rightarrow \neg(\exists\ S$ in $(CP_o + 2 \,..\, CP + SP \lhd ArrCont) \bullet$ $(SArr$ for $SP)$ matches $S)$ }

| { Variant : $Max + LP - RP - SP - CP$ }

| { Guard Negation : $CP = Max + LP - RP - SP \vee matched \vee \neg NoInterrupt$ }

od ;

if

| $(matched) \longrightarrow rep := \text{"OK"}$

$\square$

| $(\neg matched) \longrightarrow$

| $\quad$ { $NoInterrupt \Rightarrow$

| $\quad\quad CP + SP = Max + LP - RP$

| $\quad\quad \neg(\exists\ S$ in $(ArrCont$ after $CP_o + 1) \bullet (SArr$ for $SP)$ matches $S)$ }

| $\quad CP := prevCP\,;\ rep := \text{"String not found"}$

fi

```
SP := 0 : 0 .. QMax ;
Docptr := 0 : 0 .. Max ;
lastmatchOK := true : B ;
do
| (SP ≠ SP ∧ CP + Docptr < Max + LP − RP − SP ∧ lastmatchOK) →
|              lastmatchOK := CharMatched( GetArrCont( CP + Docptr + 1 )) ;
|              SP++ ; Docptr++
|   { Invariant : Docptr = matchedlength (SArr for SP, ArrCont after CP) }
|   { Invariant : lastmatchOK ⇒
|                    (SArr for SP) matches (CP .. CP + Docptr − 1 ↿ ArrCont) }
|   { Variant : SP − SP }
|   { Guard Negation : SP = SP ∨ CP + Docptr ≥ Max + LP − RP − SP ∨ ¬lastmatchOK }
od ;
if
| (SP = SP ∧ lastmatchOK)  →
|          { (SArr for SP) matches (ArrCont after CP) }
|          matched := true ; MatchedLength := Docptr
|          { MatchedLength = matchedlength(SArr for SP, ArrCont after CP) }
[]
| (SP ≠ SP ∨ ¬ lastmatchOK) →
|          { ¬ ((SArr for SP) matches (ArrCont after CP)) }
|              Skip
fi


CharMatched(c, SP)        c? : Char ; SP : 0 .. QMax

if
| (SArr(SP + 1) = . ) → MatchAll(c, SP)
[]
| (SArr(SP + 1) = \ ) → MatchEscape(c, SP)
[]
| (SArr(SP + 1) = ^ ) → MatchNot(c, SP)
[]
| (SArr(SP + 1) = [ ) → MatchRegExp(c, SP)
[]
| (SArr(SP + 1) ≠ .  ∧ SArr(SP + 1) ≠ \  ∧
|  SArr(SP + 1) ≠ ^  ∧ SArr(SP + 1) ≠ [ ) → MatchChar(c, SP)
fi
```

$MatchNot(c, SP) \qquad c? : Char \; ; \; SP : 0 \,..\, QMax$

$$\{ (SArr \text{ for } SP_o) \text{ matches } (CP \,..\, CP + Docptr - 1 \upharpoonleft ArrCont) \}$$
$$\{ SArr(SP_o + 1) = \grave{} \}$$
$$\{ c = ArrCont(CP + Docptr) \}$$
$$SP{+}{+} \; ; \; return(SP \neq SP \wedge SArr(SP + 1) \neq c)$$
$$\{ (SP \neq SP \wedge SArr(SP + 1) \neq c) \Rightarrow$$
$$(SArr \text{ for } SP + 1) \text{ matches } (CP \,..\, CP + Docptr \upharpoonleft ArrCont) \}$$

Notes

$DownSearch_{Doc8}$:

The code for *CopyQBuffSBuff* is similar to that for *CopyMTextPBuff* (Section 5.3).

$DownSearch_{Doc8}.1$:

We assume that the boolean *NoInterrupt* indicates whether or not an interrupt has occurred. The first loop invariant holds initially since *matched* is initially set to **false**.

CheckForMatch:

The loop invariant initially holds since SP is initially zero, and we appeal to the definition of "matches" and "matchedlength".

MatchNot:

The final assertion follows directly from the definition of "matches", and the other four disjuncts of *CharMatched* similarly follow.

7 Refinement Of Doc9

7.1 The Design Decision

The abstract *Doc9* state incorporates a single moveable window onto the document, achieved by embellishing *Doc8* to include the sequence of lines *WindowLines*, the window offsets *OffsetX* and *OffsetY*, and the window cursor positions *WinCurX* and *WinCurY*. In the concrete state the first and last are represented by a window on the terminal screen together with a cursor; both are provided by the hardware on which the editor is to run:

$$TerminalCursor \;\widehat{=}\; [\; TermCurX, TermCurY : \mathbb{N} \;]$$

TerminalWindow

$$TermWinLines : 1 \,..\, WinHeight \rightarrow (1 \,..\, WinWidth \rightarrow Char)$$

Note that we do not make the assumption that the terminal cursor must necessarily reside in the terminal window (to enable more realistic assumptions of the operations that the terminal provides - see Section 7.2.1). However since the abstract state requires that the window cursor resides in the terminal window, the *ConcDoc9* state must include the invariant:

$$TermCurX \in 1 \,..\, WinWidth \;\wedge\; TermCurY \in 1 \,..\, WinHeight$$

We represent the window offsets by:

$$ConcWinOffset \mathrel{\widehat{=}} [\; WinOffX, WinOffY : 0 \,..\, Max \;]$$

We also include the pointer *WinStartln* in the design: it points to the newline character immediately preceeding the document line that provides the top window line (or to zero if it is the first document line) in a similar way to that in which *Startln* identifies the cursor line; there will be *WinOffY* newlines up to *WinStartln* (and so when *WinOffY* is zero, *WinStartln* will point to the beginning of the document), and if *ptr* points to the (newline preceeding) the y^{th} window line, we have:

$$\mathsf{NumNLin}\;(ArrCont, WinStartln + 1 \,..\, ptr) + 1 = y$$

We include *WinStartln* to obviate the need for the continual re-establishment of the position in the document which will provide the starting point for the window "from scratch", which, for a long document when the window is being moved near the bottom, may be time-consuming.

We require that the terminal window correctly displays the appropriate part of the document, and so we need to define a relation which holds when a window line correctly displays a document line (with appropriate offsets), and thus we need to be able to extract a particular display line from *ArrCont*. We first define a function which takes a character array and a positive integer n as parameters, and returns the sequence of characters corresponding to the n^{th} line of the array:

$$\mathsf{ArrayLine} : (1 \,..\, Max \rightarrow Char) \times \mathbb{N}_1 \rightarrow DispLine$$

$\mathsf{ArrayLine}\;(array, y) = startlnptr + 1 \,..\, endlnptr \upharpoonleft array$
where
- $y \leq \mathsf{TotalNLin}\;(array) + 1$
- $y = \mathsf{NumNLin}\;(array, 1 \,..\, startlnptr) + 1$
- $startlnptr \leq endlnptr$
- $startlnptr \neq 0 \;\Rightarrow\; array\; startlnptr = nl$
- $endlnptr \neq \#\, array \;\Rightarrow\; array\,(endlnptr + 1) = nl$
- $\mathsf{NoNLin}\;(array, startlnptr + 1 \,..\, endlnptr)$

$\vee$
- $y > \mathsf{TotalNLin}\;(array) + 1$
- $startlnptr = endlnptr$

Thus *startlnptr* and *endlnptr* identify the y^{th} line of *array* the same way that *Startln* and *Endln* identify the $CurY^{th}$ line of *ArrCont*. Note that when the parameter y exceeds the number of lines present, the empty line is returned (when *startlnptr* equals *endlnptr*): if the window display extends beyond the bottom of the document we wish to display empty lines, and in such cases we utilise this property of ArrayLine in displaying the window.

We have the following results:

Lemma 4 : 7.1a

$$ConcDoc2$$
$$\vdash$$
$$\forall\ y : 1 \mathrel{..} DocNL + 1 \bullet \mathsf{ArrayLine}\,(ArrCont, y) = (\mathsf{FDL}^{-1}\, ArrCont)\ y$$

■

Proof

Follows directly from the definitions of ArrayLine, FDL and Lemmas 3:2.2b

■

Lemma 4 : 7.1b

$$ConcDoc2$$
$$\vdash$$
$$StartIn + 1 \mathrel{..} EndIn \upharpoonleft ArrCont = \mathsf{ArrayLine}\,(ArrCont, CurY)$$

■

Proof

Follows immediately from the *ConcDoc2* invariant, with *y* equal to *CurY*, and *startlnptr* and *endlnptr* equal to *StartIn* and *EndIn* respectively.

■

We now define a relation between a line of *ArrCont* and a *TermWinLine*, such that the former is displayed by the latter; if the array line is not equal to the *WinWidth* the window line is padded with an (invisible) null character:

$$nullchar : Char$$

$$\_\ \mathsf{isdisplayedas}\ \_ : Line \times Line \longrightarrow \mathbf{B}$$

$$\begin{aligned} &arrcontline\ \mathsf{isdisplayedas}\ termwinline \Leftrightarrow \\ &\quad (arrcontline\ \mathsf{for}\ WinWidth)\ \mathsf{prefix}\ termwinline \\ &\quad \mathsf{rng}\ (termwinline\ \mathsf{after}\ \#\, arrcontline) \subseteq \{nullchar\} \end{aligned}$$

The horizontal offset means that each document line will be displayed starting from the $(WinOffX + 1)^{st}$ position, and the following are a direct result of the definitions of "isdisplayedas" and "after":

Lemma 4 : 7.1c

$arrcontline : Line \quad | \quad \# arrcontline \leq WinOffX$

$termwinline : Line \quad | \quad \# termwinline = WinWidth$

$\mathsf{rng}\ termwinline = \{nullchar\}$

$\vdash$

$(arrcontline\ \mathsf{after}\ WinOffX)\ \mathsf{isdisplayedas}\ termwinline$

■

Lemma 4 : 7.1d

$arrcontline : Line \quad | \quad WinOffX < \# arrcontline < WinOffX + WinWidth$

$termwinline : Line \quad | \quad \# termwinline = WinWidth$

$(arrcontline\ \mathsf{after}\ WinOffX) = termwinline\ \mathsf{for}\ \# arrcontline - WinOffX$

$\mathsf{rng}\ (termwinline\ \mathsf{after}\ \# arrcontline - WinOffX) = \{nullchar\}$

$\vdash$

$(arrcontline\ \mathsf{after}\ WinOffX)\ \mathsf{isdisplayedas}\ termwinline$

■

Lemma 4 : 7.1e

$arrcontline : Line \quad | \quad \# arrcontline \geq WinOffX + WinWidth$

$termwinline : Line \quad | \quad \# termwinline = WinWidth$

$((arrcontline\ \mathsf{after}\ WinOffX)\ \mathsf{for}\ WinWidth) = termwinline$

$\vdash$

$(arrcontline\ \mathsf{after}\ WinOffX)\ \mathsf{isdisplayedas}\ termwinline$

■

We note that the $(WinOffY + y)^{th}$ line of *ArrCont* starting at position *WinOffX*, will correspond to the y^{th} window line and we are now able to define a relation which holds when an appropriate document line is correctly displayed on the terminal window:

$\mathsf{Displayed} : (0 \mathrel{..} Max \rightarrow Char) \times \mathbf{N} \times \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$

$\mathsf{Displayed}\ (array, y, OffY, OffX) \Leftrightarrow$
$\quad (\mathsf{ArrayLine}\ (array, OffY + y)\ \mathsf{after}\ OffX)\ \mathsf{isdisplayedas}\ (TermWinLines\ y)$

$\mathsf{DisplayedRange} : (0 \mathrel{..} Max \rightarrow Char) \times \mathbf{P}\,\mathbf{N} \times \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$

$\mathsf{DisplayedRange}\ (array, y1 \mathrel{..} y2, OffY, OffX) \Leftrightarrow$
$\quad \forall\ y : y1 \mathrel{..} y2 \bullet \mathsf{Displayed}\ (array, y, OffY, OffX)$

Note that we supply the array as a parameter, since this will then enable us to make assertions about the old and new *ArrCont* values in the promotion of *Doc8* operations; we use *OffX* and *OffY* rather than *OffsetX* and *OffsetY* to enable us to state, for example, that the first *y* lines of the window need scrolling up by one line:

$$\mathsf{DisplayedRange}\ (ArrCont, 1 \ldots y, WinOffX, WinOffY - 1)$$

We incorporate the boolean array *WinLineOK* of length *WinHeight* in the design, indicating whether or not each window line correctly displays the appropriate document line. Ideally we require that after each operation the window dispay is corrected, and so each entry in *WindowLineOK* is **true**; however we wish to implement the window display routines such that the display of the window may be interrupted (so that, for example, a command may be effected immediately, rather than having to wait for the window display to complete), and so we do not include this requirement in the represenation of the *Doc9* state, but require that if a particular *WindowLineOK* entry is **true**, the that window line must correctly display the corresponding document line:

ConcDoc9

ConcDoc8
TerminalDisplay
ConcWinOffset
$WinStartln : 0 \ldots Max$
$WinLineOK : 1 \ldots WinHeight \longrightarrow \mathbb{B}$
$\mathsf{NumNLin}\ (ArrCont, 1 \ldots WinStartln) = WinOffY$
$WinStartln \neq 0 \Rightarrow ArrCont\ WinStartln = nl$
$TermCurX = CurX - WinOffX \in 1 \ldots WinWidth$
$TermCurY = CurY - WinOffY \in 1 \ldots WinHeight$
$\forall\ y : 1 \ldots WinHeight \bullet$
$\quad WinLineOK\ y \Rightarrow \mathsf{Displayed}\ (ArrCont, y, WinOffX, WinOffY)$

Note that when *y* exceeds the number of lines in *ArrCont* - when the window display extends beyond the bottom of the document - the definition of **ArrayLine** ensures that empty lines are displayed (i.e. lines of *nullchar*).

The following results are a consequence of the *ConcDoc9* invariant; firstly, since the window must contain the cursor line, we show that the pointer to the window cannot exceed the pointer to the current line:

Lemma 4 : 7.1f

ConcDoc9

$\vdash$

$WinStartln \leq Startln$

■

Proof

1.	$WinLine > Startln$	assumption
2.	$ArrCont\ WinStartln = nl$	1., *ConcDoc9*
3.	$\mathsf{NumNLin}\ (ArrCont, Startln + 1 \mathinner{..} WinStartln) > 0$	1., 2.
4.	$\mathsf{NumNLin}\ (ArrCont, 1 \mathinner{..} WinStartln) = WinOffY$	*ConcDoc9*
5.	$\mathsf{NumNLin}\ (ArrCont, 1 \mathinner{..} Startln) = CurY - 1$	*ConcDoc2*
6.	$WinOffY - CurY + 1 > 0$	3., 4., 5.
7.	$CurY - WinOffY \geq 1$	*ConcDoc9*
8.	$WinStartln \leq Startln$	1., 6., 7.

■

Secondly, the y^{th} window line is provided by the $(WinOffY + y)^{th}$ document line:

Lemma 4 : 7.1g

$ConcDoc9$

$startlnptr : WinStartln \mathinner{..} Max + LP - RP$

$startlnptr \neq 0 \;\Rightarrow\; ArrCont\ startlnptr = nl$

$\mathsf{NumNLin}\ (ArrCont, 1 \mathinner{..} startlnptr) + 1 = WinOffY + y$

$\vdash$

$\mathsf{NumNLin}\ (ArrCont, WinStartln + 1 \mathinner{..} winstartlnptr) + 1 = y$

■

Proof

Follows from the definition of *WinStartln*.

■

As a result, the cursor line must appear in the $(CurY - WinOffY)^{th}$ window line:

Corollary 4 : 7.1h

$ConcDoc9$

$\vdash$

$\mathsf{NumNLin}\ (ArrCont, WinStartln + 1 \mathinner{..} Startln) + 1 = CurY - WinOffY$

■

Further, if the cursor is in the top window line, the pointer to the window and that to the current line must agree, and vice-versa:

Corollary 4 : 7.1i

$ConcDoc9$

$\vdash$

$(CurY - WinOffY = 1) \Leftrightarrow (WinStartln = Startln)$

■

Finally, when the window extends beyond the unbounded display of the document, all window lines below the document are displayed as lines of *nullchar*:

Lemma 4 : 7.1j

$ConcDoc9 \mid DocNL + 1 < WinOffY + WinHeight$

$y : DocNL + 1 - WinOffY \mathrel{..} WinHeight$

$\vdash$

$\mathsf{rng}\,(TermWinLines\ y) = \{nullchar\}$

■

Proof

By definition of $DocNL$ and ArrayLine, we have

$\mathsf{ArrayLine}\,(ArrCont, y) = \langle\rangle$

and so by the $ConcDoc9$ invariant:

$\langle\rangle \ \mathsf{isdisplayedas}\ (TermWinLines\ y)$

The definition of **isdisplayedas** now provides the result.

■

7.1.1 The Concrete-Abstract Invariant

Rel_{Doc9}

$Doc9$ $ConcDoc9$ Rel_{Doc8}
$OffsetX, OffsetY = WinOffX, WinOffY$ $WinCurX, WinCurY = CurX - WinOffX, CurY - WinOffY$ $WindowLines = TermWinLines$

Clearly, the schema uniquely defines each abstract window component for a given concrete state, and we establish:

$$Rel_{Doc9}$$

$\vdash$

$$\forall\ ConcDoc9 \bullet \exists_1\ Doc9 \bullet Rel_{Doc9}$$

■

We calculate:

$$AbsRel_{Doc9} \mathrel{\hat{=}} \Xi Doc9 \land AbsRel_{Doc8}$$

and, as in section 4.1.1, obtain:

Lemma 4 : 7.1.1b

$\vdash$

$$Doc9 \sqsubseteq_{\infty} ConcDoe9$$

■

We finally calculate *concRel*:

$$
\begin{array}{|l}
ConcRel_{Doc9} \\
\hline
\Delta ConcDoc9 \\
ConcRel_{Doc8} \\
\hline
WinOffX', WinOffY' = WinOffX, WinOffY \\
WinStartln' = WinStartln \\
TermWinLines' = TermWinLines \\
TermCurX', TermCurY' = TermCurX, TermCurY \\
\hline
\end{array}
$$

Similar comments to those made in Section 4.1.1 also apply here, allowing use to use $Standardize_{Doc1}$ as the concrete reorganizing operation.

7.2 Displaying The Document On The Terminal Window

We refine the *Doc9* operations in such a way that the array *WinLineOK* correctly indicates which window lines are currently correctly displayed (to conform to the *ConcDoc9* invariant), but do not re-display incorrectly displayed lines; such lines set to **false** in the *WindowLineOK* array, and are displayed by the main program loop when there is currently no command to be processed and no user interrupt pending (see Section 6.3).

However we do employ window scrolling when at least half of the current window can be moved to its correct position (see Section 7.2.4), and assume that the terminal hardware is provided with scrolling and other basic window display operations; we state our assumptions of these operations in the next section.

7.2.1 Specifications Of Operations Provided By The Terminal

We assume implementation-dependent operations to clear the window (requiring no argument, filling the window with *nullchar* and homing the cursor), to set the cursor (taking two integer arguments), to display a character at the terminal cursor position (taking a single character argument and incrementing the cursor as appropriate) and to clear the terminal cursor line from the cursor position (filling the rest of the line with *nullchar*). The last two operations require that the cursor is positioned within the terminal screen, and although we specify them as total operations, we do not specify what happens to the window display when the cursor is incorrectly positioned, or to the cursor position when the operation takes it outside the window.

The operation to clear the screen is assumed to fill the screen with *nullchar*, and moves the cursor to the top left hand corner of the screen:

CLS

$\Delta TerminalDisplay$

$TermCurX', TermCurY' = 1, 1$
$\forall\ y : 1 \,..\, WinHeight \bullet \mathsf{rng}\,(TermWinLines'\ y) = \{nullchar\}$

We refine this operation to additionally set each entry in the *WinLineOK* array to false [$\sqsubseteq$ 2 : 3.2*a*]:

CLSAdjust

$CLS\ ;\ WinLinesBad(1, WinHeight)$

WinLinesBad $\qquad first?, last? : 1 \,..\, WinHeight$

if
$\quad (first < 1) \longrightarrow first := 1$
$\square$
$\quad (last > WinHeight) \longrightarrow last := WinHeight$
$\square$
$\quad (first \geq 1 \wedge last \leq WinHeight) \longrightarrow Skip$
fi ;
do
$\quad (first \leq last) \longrightarrow WinLineOK\ first := \mathsf{false}\ ;\ \mathsf{first}{+}{+}$
$\quad$ **{** Invariant : $\forall\ i : first_o \,..\, first - 1 \bullet \neg\,(WinLineOK\ i)$ **}**
$\quad$ **{** Variant : $last - first$ **}**
$\quad$ **{** Guard Negation : $first - 1 = last$ **}**
od

The operation to set the terminal cursor does not otherwise alter the display:

$$
\begin{array}{l}
\textit{SetTermCursor} \\
\hline
\Delta \textit{TerminalDisplay} \\
x?, y? : \mathbb{N} \\
\hline
\textit{TermCurX}', \textit{TermCurY}' = x?, y? \\
\textit{TermWinLines}' = \textit{TermWinLines} \\
\hline
\end{array}
$$

Provided the cursor is in the screen, the operation to display a character leaves the screen otherwise unchanged, and moves the cursor one position to the right:

$$
\begin{array}{l}
\textit{DisplayChar} \\
\hline
\Delta \textit{TerminalDisplay} \\
c? : \textit{Char} \\
\hline
\quad \textit{TermCurX} \in 1 \,..\, \textit{WinWidth} \ \wedge\ \textit{TermCurY} \in 1 \,..\, \textit{WinHeight} \\
\quad \{\textit{TermCurY}\} \lhd \textit{TermWinLines}' = \{\textit{TermCurY}\} \lhd \textit{TermWinLines} \\
\quad \textit{TermWinLines}'\ \textit{TermCurY} = \\
\qquad\qquad \textit{TermWinLines}\ \textit{TermCurY} \oplus \{\textit{TermCurX} \mapsto c?\} \\
\quad \textit{TermCurX}', \textit{TermCurY}' = \textit{TermCurX} + 1, \textit{TermCurY} \\
\vee \\
\quad \textit{TermCurX} \notin 1 \,..\, \textit{WinWidth} \ \vee\ \textit{TermCurY} \notin 1 \,..\, \textit{WinHeight} \\
\hline
\end{array}
$$

The operation to clear to the end of the line fills the cursor line from the cursor position with *nullchar*. The cursor line must initially be within the window range; we make no assumption about the final position of the cursor:

$$
\begin{array}{l}
\textit{ClearToEndOfLine} \\
\hline
\Delta \textit{TerminalDisplay} \\
\hline
\quad \textit{TermCurY} \in 1 \,..\, \textit{WinHeight} \\
\quad \{\textit{TermCurY}\} \lhd \textit{TermWinLines}' = \{\textit{TermCurY}\} \lhd \textit{TermWinLines} \\
\quad ((\textit{TermWinLines}'\ \textit{TermCurY}) \ \mathsf{for}\ \textit{TermCurX} - 1) = \\
\qquad\qquad ((\textit{TermWinLines}'\ \textit{TermCurY}) \ \mathsf{for}\ \textit{TermCurX} - 1) \\
\quad \mathsf{rng}\ ((\textit{TermWinLines}'\ \textit{TermCurY}) \ \mathsf{after}\ \textit{TermCurX} - 1) \subseteq \{\textit{nullchar}\} \\
\vee \\
\quad \textit{TermCurY} \notin 1 \,..\, \textit{WinHeight} \\
\hline
\end{array}
$$

We also assume the existence of operations to scroll the display up and down n lines; the former scrolls the entire screen from the bottom line and introduces n lines of *nullchar* at the bottom (typically achieved by continued re-positioning of the cursor at the start of the bottom line and printing a newline character), and the latter takes a second argument indicating from which line the scroll is to take place, with n lines of null char being introduced from that line (typically achieved by continued re-positioning the cursor at the start of the parameter line and adding a blank line):

ScrollUp

$$
\begin{array}{|l}
\Delta \mathit{TerminalWindow} \\
n? : \mathbb{N} \\
\hline
\quad n? \in 1 \mathrel{..} \mathit{WinHeight} \\
\quad \forall\ y : 1 \mathrel{..} \mathit{WinHeight} - n? \bullet \\
\qquad\qquad \mathit{TermWinLines}'\ y = \mathit{TermWinLines}(y + n?) \\
\quad \forall\ y : \mathit{WinHeight} - n? + 1 \mathrel{..} \mathit{WinHeight} \bullet \\
\qquad\qquad \mathrm{rng}\ (\mathit{TermWinLines}'\ y) = \{\mathit{nullchar}\} \\
\vee \\
\quad n? \notin 1 \mathrel{..} \mathit{WinHeight}
\end{array}
$$

In order to preserve the *ConcDoc9* invariant, we must ensure the update of the *WinLineOK* array by setting each entry in the range 1 to $(\mathit{WinHeight} - n)$ to that of the y^{th} entry below it, and that the final n entries are set to **false** (since the last n screen lines will consist of *nullchar*, irrespective of the document content). We therefore refine the operation, to *ScrollUpAdjust*, as follows:

ScrollUpAdjust(n) $\qquad n? : \mathbb{N}_+$

$$
\begin{array}{l}
\{\ n \in 1 \mathrel{..} \mathit{WinHeight}\ \} \\
y := 0 : 1 \mathrel{..} \mathit{WinHeight}\ ; \\
\mathbf{do} \\
\quad (y + n < \mathit{WinHeight}) \longrightarrow \mathit{WinLineOK}(y+1) := \mathit{WinLineOK}(y + n + 1)\ ;\ y{+}{+}\ ; \\
\quad \{\ \textsf{Invariant} : \forall\ i : 1 \mathrel{..} y \bullet \mathit{WinLineOK}\ i = \mathit{WinLineOK}_o(i + n)\ \} \\
\quad \{\ \textsf{Variant} : \mathit{Winheight} - y - n\ \} \\
\quad \{\ \textsf{Guard Negation} : y = \mathit{WinHeight} - n\ \} \\
\mathbf{od}\ ; \\
\{\ \mathit{WinLinesBad}(\mathit{WinHeight} - n + 1, \mathit{WinHeight})\ \}
\end{array}
$$

ScrollDown

$$
\begin{array}{|l}
\Delta \mathit{TerminalWindow} \\
n?, \mathit{winline}? : \mathbb{N} \\
\hline
\quad \mathit{winline}? \in 1 \mathrel{..} \mathit{WinHeight} \wedge n? \leq \mathit{WinHeight} - \mathit{winline}? + 1 \\
\quad \forall\ y : 1 \mathrel{..} \mathit{winline}? - 1 \bullet \\
\qquad\qquad \mathit{TermWinLines}'\ y = \mathit{TermWinLines}\ y \\
\quad \forall\ y : \mathit{winline}? \mathrel{..} \mathit{winline}? + n? - 1 \bullet \\
\qquad\qquad \mathrm{rng}\ (\mathit{TermWinLines}'\ y) = \{\mathit{nullchar}\} \\
\quad \forall\ y : \mathit{winline} + n? \mathrel{..} \mathit{WinHeight} \bullet \\
\qquad\qquad \mathit{TermWinLines}'\ y = \mathit{TermWinLines}(y - n?) \\
\vee \\
\quad \mathit{winline}? \notin 1 \mathrel{..} \mathit{WinHeight} \vee n? > \mathit{WinHeight} - \mathit{winline}? + 1
\end{array}
$$

We refine this operation to *ScrollDownAdjust* in a similar way to that in which we refine *ScrollUp*.

7.2.2 An Operation To Set "WinStartln"

We use the result of Lemma 4:7.1h that there are $(CurY - WinOffY - 1)$ newlines in *ArrCont* between the values of *WinStartln* and *Startln*.

SetWinStartln

$numnl := CurY - WinOffY - 1 : 1 \,..\, WinHeight$;

$WinStartln := Startln$;

do

| $(numnl \neq 0) \longrightarrow WinStartln := GetPrevStartlnptr(WinStartln)$; $numnl--$

| { Invariant : $WinStartln \neq 0 \Rightarrow ArrCont\ WinStartln = nl$ }

| { Invariant : $numnl \neq 0 \Rightarrow WinStartln \neq 0$ }

| { Invariant : $\mathsf{NumNLin}\ (ArrCont, WinStartln + 1 \,..\, Startln) =$
| $CurY - WinOffY - 1 - numnl$ }

| { Variant : $numnl$ }

| { Guard Negation : $numnl = 0$ }

od

Note
The code for *GetPrevStartlnptr* is similar to that for *GetNextStartlnptr*, given in Section 7.2.3, and the second invariant is due to Lemma 4:7.1h. We appeal to [$\sqsubseteq 2 : 3.5.3a$] for the loop.

7.2.3 An Operation To Display The Window

Displaying A Window Line

We first give an operation that displays the $(WinOffY + y)^{th}$ line of the document on the y^{th} window line: the operation's parameter, *startlnptr* points to the newline immediately preceeding the start of the $(WinOffY + y)^{th}$ document line (or to zero if it is the first document line). The first loop, *MoveWinOffX*, moves the pointer *WinOffX* positions along the line (length permitting), and the second loop, *DisplayFromWinOffX*, displays the next *WinWidth* characters of the line (again, length permitting); if *WinWidth* characters have not been displayed, determined by *ClearToEndOfLine*, the *ClearToEndOfLine* operation is effected.

Code for $DisplayWindowLine(startInptr)$ $\qquad startInptr? : 0 .. Max + LP - RP$;

$x := 0 : 0 .. Max$;

$\{ startInptr \neq Max + LP - RP \Rightarrow$ $\mathsf{NumNLin}\ (ArrCont, 1 .. startInptr) + 1 = WinOffY + y \}$

$\{ 0 \neq startInptr \neq Max + LP - RP \Rightarrow ArrCont\ startInptr = nl \}$

$\{ TermCurX, TermCurY = 1, y \}$

$\{ arrcontline = \mathsf{ArrayLine}(ArrCont, WinOffY + y) \wedge termwinline = TermWinLine\ y \}$

$MoveWinOffX$;

$\{ n = x \}$

$DisplayFromWinOffX$;

$\{ x < WinOffX + WinWidth \Rightarrow \#\, arrcontline = x \}$

$\{ x = WinOffX + WinWidth \Rightarrow n = WinOffX \wedge \#\, arrcontline \geq x \}$

$ClearToEndOfLine$

$\{ \mathsf{Displayed}\ (ArrCont, y, WinOffX, WinOffY) \}$

$MoveWinOffX$

do

$(startInptr + x \neq Max + LP - RP \wedge$
$GetArrCont(startInptr + x + 1) \neq nl \wedge x \neq WinOffX) \longrightarrow x{+}{+}$

$\{ \mathsf{Invariant:}\ (startInptr + 1 .. startInptr + x \upharpoonleft ArrCont)\ \mathsf{prefix}\ arrcontline \}$

$\{ \mathsf{Variant:}\ Max + LP - RP - x \}$

$\{ \mathsf{Guard\ Negation:}\ (startInptr + x = Max + LP - RP \vee$
$GetArrCont(startInptr + x + 1) = nl \vee x = WinOffX) \}$

od

$DisplayFromWinOffX$

do

$(startInptr + x \neq Max + LP - RP \wedge$
$GetArrCont(startInptr + x + 1) \neq nl \wedge x \neq WinOffX + WinWidth) \longrightarrow$
$DisplayChar(GetArrCont(startInptr + x + 1))$; $x{+}{+}$

$\{ \mathsf{Invariant:}\ (startInptr + 1 .. startInptr + x \upharpoonleft ArrCont)\ \mathsf{prefix}\ arrcontline \}$

$\{ \mathsf{Invariant:}\ ((arrcontline\ \mathsf{after}\ n)\ \mathsf{for}\ x - n) = termwinline\ \mathsf{for}\ x - n \}$

$\{ \mathsf{Invariant:}\ TermCurX = x - n + 1 \}$

$\{ \mathsf{Variant:}\ Max + LP - RP - x \}$

$\{ \mathsf{Guard\ Negation:}\ (startInptr + x = Max + LP - RP \vee$
$GetArrCont(startInptr + x + 1) = nl \vee x = WinOffX + WinWidth) \}$

od

Note

The *ConcDoc9* invariant stipulates that when the terminal display extends beyond the bottom of the document, such lines should contain *nullchar* (Lemma 4:7.1j). When the parameter passed is ($Max + LP - RP$) (i.e. the bottom of the document will have been reached), neither loop will start. If the parameter passed is not ($Max + LP - RP$), the definition of ArrayLine stipulates that there must be ($WinOffY + y - 1$) newlines up to the parameter (and hence the first assertion for the operation) in order to ensure that the correct document line is displayed.

The invariant for *MoveWinOffX* and the first for *DisplayFromWinOffX* are due to the definition of ArrContLine; the remaining two invariants for the latter are provided by *DisplayChar*. We are able to make the assertions immediately after *DisplayFromWinOffX* from the guards of both loops, together with the definition of ArrayLine. Our final assertion that the window line displays the array line is due to Lemmas 4:7.1c, 4:7.1d and 4:7.1e.

Displaying The Cursor Line

Many operations will necessitate the re-display of just the cursor line (for example a newline insert), and in this case the parameter for the operation will be *Startln*:

Code for *DisplayCurLine*

{ NumNLin ($ArrCont, Startln$) = $CurY - 1$ }
$SetTermCursor(1, CurY - WinOffY)$
$DisplayWindowLine(Startln)$
{ Displayed ($ArrCont, TermCurY, WinOffX, WinOffY$) }

Displaying A Range Of Window Lines

We next wish to define an operation that will display a range of window lines by repeatedly calling *DisplayWindowLine* with appropriate *startlnptr* parameters; we first define an operation which increments that parameter; the maximum value of the parameter is ($Max + LP - RP$) (when the display of the document will be complete, but the display of the window is not), and if that maximum value is given as the parameter, that same value is returned:

Code for *GetNextStartInptr*(*ptr*) $\quad ptr? : 0 \,..\, Max$;

$\{\, 0 \neq ptr \neq Max + LP - RP \;\Rightarrow\; ArrCont\ ptr = nl \,\}$

if

$(ptr \neq Max + LP - RP) \longrightarrow$

$ptr{+}{+}$;

do

$(ptr \neq Max + LP - RP \;\wedge\; GetArrCont\ ptr \neq nl) \longrightarrow ptr{+}{+}$

$\{$ Invariant : NoNLin $(ArrCont, ptr_o + 1 \,..\, ptr - 1)\,\}$

$\{$ Variant : $Max + LP - RP - ptr\,\}$

$\{$ Guard Negation : $ptr \neq Max + LP - RP \;\Rightarrow\; ArrCont\ ptr = nl\,\}$

od

▯

$(ptr = Max + LP - RP) \longrightarrow Skip$

fi ;

return(*ptr*)

$\{\, ptr_o = Max + LP - RP \;\Rightarrow\; ptr = Max + LP - RP \,\}$

$\{\, ptr \neq Max + LP - RP \;\Rightarrow$

$ArrCont\ ptr = nl$

NumNLin $(ArrCont, 1 \,..\, ptr)$ = NumNLin $(ArrCont, 1 \,..\, ptr_o) + 1\,\}$

We now define the operation which displays a range of window lines:

Code for $DisplayWindowRange(first, last)$ $\quad first?, last? : 1 \,..\, WinHeight$;

$ptr := WinStartln : 0 \,..\, Max$; $\quad y := first : 1 \,..\, WinHeight$;
{ NumNLin $(ArrCont, 1 \,..\, ptr) = WinOffY$ }
do
$(y \neq 1) \longrightarrow ptr := GetNextStartln(ptr)$; $y--$
{ Invariant : $0 \neq ptr \neq Max + LP - RP \Rightarrow ArrCont\ ptr = nl$ }
{ Invariant : $ptr \neq Max + LP - RP \Rightarrow$
NumNLin $(ArrCont, 1 \,..\, ptr) = WinOffY + first - y$ }
{ Variant : y }
{ Guard **Negation** : $y = 1$ }
od ;
do
$(first \leq last \wedge \neg CharAvailable \wedge NoInterrupt) \longrightarrow$
$DisplayLineIfNecessary$; $ptr := GetNextStartln(ptr)$; $first++$
{ Invariant : $0 \neq ptr \neq Max + LP - RP \Rightarrow ArrCont\ ptr = nl$ }
{ Invariant : $ptr \neq Max + LP - RP \Rightarrow$
NumNLin $(ArrCont, 1 \,..\, ptr) = WinOffY + first - 1$ }
{ Invariant : DisplayedRange$(ArrCont, first_o \,..\, first - 1, WinOffX, WinOffY)$ }
{ Variant : $last - first$ }
{ Guard **Negation** : $(\neg CharAvailable \wedge NoInterrupt) \Rightarrow first - 1 = last$ }
od

DisplayLineIfNecessary

if
$\neg(WinLineOK\ first) \longrightarrow SetTermCursor(1, first)$; $DisplayWindowLine(ptr)$;
$WinLineOK(first) :=$ **true**
□
$(WinLineOK\ first) \longrightarrow Skip$
fi

Note

If the range (*first* .. *last*) is empty, the second loop guard ensures that no window lines are displayed. As discussed in Section 7.1, the display of a range of window lines may be interrupted by a command being entered at the keyboard (in which case we assume *CharAvailable* to hold) or a user-interrupt being received (in which case we assume that *NoInterrupt* is set to **false** - see Section 6.4.2).

Displaying The Window

We may now give the operation to display the entire window:

DisplayTheWindow

$$DisplayWindowRange(1,\ WinHeight)$$
$$\{\ \mathsf{DisplayedRange}(ArrCont,\ 1\ ..\ WinHeight,\ WinOffX,\ WinOffY)\ \}$$

7.2.4 An Operation To Move The Window Vertically

Some *Doc8* operations will cause the cursor to leave the current window and thus a change in either or both offsets will be necessary in order to regain the cursor. We consider the case when the window needs to be moved y positions vertically downwards to regain the cursor: this may be necessitated by a right move or left insert, and for the latter we assume that the range of window lines after *first* is incorrectly displayed by a factor of y with respect to the current offset (both values being parameters to the operation). We define an operation which produces a correct window display, with specification:

MoveWindowDown

$\Delta ConcDoc9$
$\Xi Doc8$
$y? : \mathbb{N}_1$
$first? : 1\ ..\ WinHeight$

$OPType = LeftInsert \Rightarrow$
$\quad \mathsf{DisplayedRange}(ArrCont, first?\ ..\ WinHeight,\ WinOffX,\ WinOffY - y?)$
$WinOffX',\ WinOffY' = WinOffX,\ WinOffY + y?$

Code:

$MoveWindowDown(y, first) \qquad y? : \mathbb{N}_1\ ;\ \ first? : 1\ ..\ WinHeight$

$\{\ CurX - WinOffX \in 1\ ..\ WinWidth\ \}$
$\{\ CurY - WinOffY + y \in 1\ ..\ WinHeight\ \}$
$WinOffY\ := \ WinOffY + y\ ;\ SetWinStartln\ ;$
if
$\mid (OPType = RightMove) \longrightarrow MoveWindowDown\_RightMove$
$\square$
$\mid (OPType \neq RightMove) \longrightarrow MoveWindowDown\_LeftInsert$
fi

MoveWindowDown_RightMove

$$\textbf{if} \; (y \leq HalfWinHeight + 1) \;\longrightarrow\; ScrollUpAdjust(y) \;\square\; (y > HalfWinHeight + 1) \;\longrightarrow\; WinLinesBad(1, WinHeight) \;\textbf{fi}$$

Note
We could always re-display the entire window to produce a correct display, but we take the view (for efficiency reasons) that if at least half of the current window can be scrolled into its correct position we do so, leaving the remaining part of the screen to be re-displayed; we ensure that the appropriate *WinLineOK* entries are set to **false**, forcing the re-display of such lines.

We may similarly specify the analogous operation *MoveWindowUp*, noting that the operation may be necessitated by either a left move or left delete operation.

7.3 Promotion Of The Doc8 Operations

Each of the *Doc8* operations is promoted to the *Doc9* state by post-sequential composition with *WindowPolicy*, where:

$$WindowPolicy \;\widehat{=}\; CursorInWindow \vee Scroll \vee Pan \vee ScrollAndPan$$

after which all *ConcDoc9* invariants will hold, with the exception that the terminal cursor may be incorrectly set, and we rectify by post sequential composition with:

SetDocCursor

$$SetTermCursor(CurX - WinOffX, CurY - WinOffY)$$

We note that the specification stipulates that if the *Doc8* operation moves the cursor outside the current window *both* window offsets should be changed only when either a scroll or a pan will not regain the cursor: when the operation leaves the cursor in the window the specification allows for a window change.

7.3.1 Refinement Of "Scroll"

Our scrolling policy is that for a downward scroll, the window is re-positioned such that the cursor is a quarter of the screen height from the bottom, and for an upward scroll it is positioned the same distance from the top (document length permitting), and we introduce:

$$QtrWinHeight : \mathbb{N}_1 \mid QtrWinHeight = WinHeight - PageHeight < HalfWinHeight$$

Specification:

$$
\begin{array}{|l}
\textit{Scroll} \\
\hline
\Delta WindowCursor \\
\Xi DocCursor \\
\hline
DocCurX - OffsetX \in 1 \mathrel{..} WinWidth \\
DocCurY - OffsetY \notin 1 \mathrel{..} WinHeight \\
OffsetX' = OffsetX \\
DocCurY' - OffsetY' \in 1 \mathrel{..} WinHeight \\
\hline
\end{array}
$$

Weakest concrete operation:

$$
\left[\begin{array}{l}
CurX - WinOffX \in 1 \mathrel{..} WinWidth \\
CurY - WinOffY \notin 1 \mathrel{..} WinHeight \\
WinOffX' = WinOffX \\
CurY' - WinOffY' \in 1 \mathrel{..} WinHeight
\end{array}\right.
$$

Code:

$Scroll(first, last) \qquad first?, last? : \mathbf{N}_1$

$$
\begin{array}{l}
winy := CurY - WinOffY : \mathbf{N}_1 ; \\
\{\, winy < 1 \;\lor\; winy > WinHeight \,\} \\
\textbf{if} \\
\quad (winy < 1 \;\land\; CurY \geq QtrWinHeight) \longrightarrow \\
\qquad \{\, 0 \leq CurY - QtrWinHeight \,\} \\
\qquad MoveWindowUp(\, WinOffY - CurY + QtrWinHeight, last) \\
\qquad \{\, WinOffY = CurY - QtrWinHeight \,\} \\
\qquad \{\, 0 \leq WinOffY \,\} \\
\qquad \{\, CurY - WinOffY = QtrWinHeight < WinHeight \,\} \\
\square \\
\quad (winy < 1 \;\land\; CurY < QtrWinHeight) \longrightarrow \\
\qquad MoveWindowUp(\, WinOffY, last) \\
\qquad \{\, WinOffY = 0 \,\} \\
\qquad \{\, CurY - WinOffY < QtrWinHeight < WinHeight \,\} \\
\square \\
\quad (winy > WinHeight) \longrightarrow \\
\qquad \{\, CurY > WinHeight \,\} \\
\qquad MoveWindowDown(CurY - PageHeight - WinOffY, first) \\
\qquad \{\, WinOffY = CurY - PageHeight \,\} \\
\qquad \{\, WinOffY > QtrWinHeight \geq 1 \,\} \\
\qquad \{\, CurY - WinOffY = PageHeight > 1 \,\} \\
\textbf{fi}
\end{array}
$$

Note

The *ConcDoc9* invariant means that it is necessary to show that *WinOffY* does not become negative and that, in the case of *MoveWindowUp*, (*CurY* − *WinOffY*) does not exceed *WinHeight*, and in the case of *MoveWindowDown*, it exceeds zero.

Our panning policy and, hence, the refinement of *Pan* is similar (using *QtrWinWidth* in an analogous way to *QtrWinHeight*), using the *MoveWindowLeft* and *MoveWindowRight* operations of Section 7.4, with the refinement of *ScrollAndPan* being a combination of both policies.

If an operation leaves the cursor visible, the first disjunct of *WindowPolicy* will apply:

$CursorInWindow$

$$\Delta WindowOffset$$
$$\Xi Doc8$$

$$DocCurX - OffsetX \in 1 \mathinner{..} WinWidth$$
$$DocCurY - OffsetY \in 1 \mathinner{..} WinHeight$$
$$DocCurX' - OffsetX' \in 1 \mathinner{..} WinWidth$$
$$DocCurY' - OffsetY' \in 1 \mathinner{..} WinHeight$$

We refine to the operation *CorrectDisplay*, which has many similarities with both of the operations *MoveWindowDown* and *MoveWindowUp*, taking as parameters *first* and *last* (indicating the range of lines incorrectly displayed), but not taking the parameter y, since it will not be necessary to move the window.

We may now refine *WindowPolicy* as follows:

$WindowPolicy(first, last) \qquad first?, last? : \mathbb{N}_1$

```
winx := CurX - WinOffX : 1 .. WinWidth ;
winy := CurY - WinOffY : 1 .. WinHeight ;
if
  (winx ≥ 1 ∧ winx ≤ WinWidth) →
        if
        | (winy ≥ 1 ∧ winy ≤ WinHeight) → CursorInWindow(first, last)
        []
        | (winy < 1 ∨ winy > WinHeight) → Scroll(first, last)
        fi
[]
  (winx < 1 ∨ winx > WinWidth) →
        if
        | (winy ≥ 1 ∧ winy ≤ WinHeight) → Pan
        []
        | (winy < 1 ∨ winy > WinHeight) → ScrollAndPan
        fi
fi
```

Thus for each operation *OP* defined on the *Doc8* state and in the set *CursorOps?*, we have, as the code for the corresponding *Doc9* operation:

$prevDocNL := DocNL : 0 \,..\, Max$;

$rep := OP_{Doc8} : Report$;

$first, last, temp : -Max \,..\, Max$;

if

| $(rep \neq \text{"OK"}) \longrightarrow return(rep)$

▯

| $(rep = \text{"OK"}) \longrightarrow$

  if

  | $(EState = State_{Quote}) \longrightarrow DisplayTheQuoteBuffer$; $return(\text{"OK"})$

  ▯

  | $(EState = State_{Doc}) \longrightarrow Set\_firstlast$;

  $WindowPolicy(first, last)$;

  $SetDocCursor$; $return(\text{"OK"})$

  fi

fi

Set_firstlast

$first := prevCurY - WinOffY$;

$last := first + DocNL - prevDocNL + NLRem - NLIns$;

if

| $(first > last) \longrightarrow temp := first$; $first := last$; $last := temp$

▯

| $(first \leq last) \longrightarrow Skip$

fi

Note

We establish *first* and *last* using *DocNL* and *prevDocNL* (taking account of *NLRem* and *NLIns*), rather than *CurY* and *prevCurY*, since for right delete and right insert operations, the latter method will fail. The *DisplayTheQuoteBuffer* operation is similar to *DisplayCurLine* (except that a pointer parameter is not necessary).

7.3.2 Promotion Of Content-Changing Operations

We choose to promote each content-changing operation separately since each will require different treatment, our proof obligation being to demonstrate that the new (possibly unchanged) window correctly displays the document, that it contains the new cursor position and that both new window offsets are non-zero. We promote *InsertChar* to illustrate the method.

Promotion Of "InsertChar"

We distinguish four cases: the cursor being visible after the insertion of a newline, the cursor being visible after the insertion of a character other than a newline, the cursor not being visible after the insertion of a newline, and the cursor not being visible after a non-newline insert. We note that after a newline insert, the value of *CurX* will be *1*, and the *ConcDoc9* invariant dictates that the final value of *WinOffX* be zero.

For the first of these cases we use Lemma 3:2.2a, together with the fact that the y^{th} array line will provide the $(y - WinOffY)^{th}$ window line, and, since the *Doc8* operation will have incremented *CurY* by one, the $TermCurY^{th}$ window line will correspond to the $(CurY - WinOffY - 1)^{st}$ document line, the previous cursor line. After the insertion of a newline character into window line *TermCurY*, the first $(TermCurY - 1)$ lines will remain unchanged, and the new window lines from $(TermCurY + 1)$ to *WinHeight* will correspond to those in the current window from *TermCurY* to $(WinHeight - 1)$ - i.e. these lines will be currently displayed by an offset of $(WinOffY + 1)$ (since the offset still has its original value). Since the insertion will have been at window position *TermCurX*, the new $TermCurY^{th}$ window line will be the same as the current one, but cut off after $(TermCurX - 1)$ characters (since the horizontal window offset will not have changed and must therefore have been zero), with the rest of that current window line providing the new $(TermCurY + 1)^{st}$ window line.

The operation thus consists of clearing to the end of the $TermCurY^{th}$ line, scrolling down one from the $(TermCurY + 1)^{st}$ line, and displaying the $(TermCurY + 1)^{st}$ - the new cursor - line.

CursorVisibleAndNewline

> { $TermCurX \in 1 .. WinWidth \ \wedge \ TermCurY \in 1 .. WinHeight$ }
> { $TermCurY = CurY - WinOffY - 1 \ \wedge \ CurX = 1$ }
> { $WinOffX = 0$ }
> { DisplayedRange (*ArrCont*, *1* .. *TermCurY* − *1*, *WinOffX*, *WinOffY*) }
> { DisplayedRange (*ArrCont*, *TermCurY* + *1* .. *WinHeight*, *WinOffX*, *WinOffY* + *1*) }
> { (ArrayLine (*ArrCont*, *WinOffY* + *TermCurY*)) =
> (*TermWinLines TermCurY*) for *TermCurX* − *1* }
> *ClearToEndOfLine* ;
> { rng ((*TermWinLines TermCurY*) after *TermCurX* − *1*) $\subseteq$ { *nullchar* } }
> { Displayed (*ArrCont*. *TermCurY*, *WinOffX*, *WinOffY*) }
> *ScrollDown*(*1*, *CurY* − *WinOffY*) ; *DisplayPrompt* ;
> { DisplayedRange (*ArrCont*, *TermCurY* + *2* .. *WinHeight*, *WinOffX*, *WinOffY*) }
> *DisplayCurLine*
> { DisplayedRange (*ArrCont*, *1* .. *WinHeight*, *WinOffX*, *WinOffY*) }

For the second case, the only line that will change is the cursor line from the previous cursor position; because we are unsure how many characters will have been inserted (it may have been a *tab* insert) and we don't assume a terminal operation to supply the position of the terminal cursor, we require the previous value of *CurX* to be input (*TermCurY* not having changed):

CursorVisibleAndNotNewline(*prevCurX*)   *prevCurX? : 1 .. WinWidth*

$tempCurX \; := \; CurX : 1 \,..\, WinWidth$;

{ $TermCurX \in 1 \,..\, WinWidth \;\wedge\; TermCurY \in 1 \,..\, WinHeight$ }

{ $TermCurX = prevCurX \;\wedge\; TermCurY = CurY - WinOffY$ }

{ **DisplayedRange** ($ArrCont, 1 \,..\, TermCurY - 1, WinOffX, WinOffY$) }

{ **DisplayedRange** ($ArrCont, TermCurY + 1 \,..\, WinHeight, WinOffX, WinOffY$) }

{ ((**ArrayLine** ($ArrCont, CurY$) **after** $WinOffX$) **for** $TermCurX - 1$) **prefix**

($TermWinLines\ TermCurY$) }

$CurX \; := \; prevCurX$; *DisplayCurLineFromCur* ; $CurX \; := \; tempCurX$

{ **Displayed** ($ArrCont, TermCurY, WinOffX, WinOffY$) }

{ **DisplayedRange** ($ArrCont, 1 \,..\, WinHeight, WinOffX, WinOffY$) }

For the third case, either the newline insert will necessitate a change in horizontal window offset (in which case a pan will be necessary, and the window completely re-displayed) or the newline insert was in the bottom window line. For the latter, if a pan is not necessary, the bottom window line is cleared from the previous terminal cursor position (for the same reasons as those stated in the first case): the display will then be correct, and so we may use a scroll. We may recognize the cases when a pan is necessary by *WinOffX* being non-zero (since it must be zero after the promotion); in this case we employ *WindowPolicy* which will result in the window being completely re-displayed:

CursorNotVisibleAndNewline(*prevCurX*)

{ $TermCurX \in 1 \,..\, WinWidth \;\wedge\; TermCurY \in 1 \,..\, WinHeight$ }

{ $TermCurY = CurY - WinOffY - 1 \;\wedge\; CurX = 1$ }

if

| ($WinOffX = 0$) $\longrightarrow$

  { $TermCurY = WinHeight$ }

  { **DisplayedRange** ($ArrCont, 1 \,..\, WinHeight - 1, WinOffX, WinOffY$) }

  { (**ArrayLine** ($ArrCont, WinOffY + WinHeight$) **after** $WinOffX$) =

  ($TermWinLines\ WinHeight$) **for** $TermCurX - 1$ }

  ClearToEndOfLine ;

  { **Displayed** ($ArrCont, WinHeight, WinOffX, WinOffY$) }

  { **DisplayedRange** ($ArrCont, 1 \,..\, WinHeight, WinOffX, WinOffY$) }

  Scroll

▯

| ($WinOffX \neq 0$) $\longrightarrow$ { $CurX - WinOffX < 1$ }

  WindowPolicy

fi

{ **DisplayedRange** ($ArrCont, 1 \,..\, WinHeight, WinOffX, WinOffY$) }

For the last case, since the cursor line will not have changed, ($CurX - WinOffX$) must exceed *WinWidth* after the *Doc8* operation (since *CurX* will have increased), and so we employ *Pan*, resulting in a complete re-display of the screen.

We now have:

$InsertChar_{Docs9}(c)$ $\qquad c? : Char$

$prevCurX := CurX : 1 \,..\, WinWidth$;
$rep := InsertChar(c)_{Docs8} : Report$;
if
$\mid (rep \neq \text{"OK"}) \longrightarrow return(rep)$
$\Box$
$\mid (rep = \text{"OK"}) \longrightarrow$
  if
  $\mid (CursorVisible \wedge c = nl) \longrightarrow CursorVisibleAndNewLine$
  $\Box$
  $\mid (CursorVisible \wedge c \neq nl) \longrightarrow CursorVisibleAndNotNewLine$
  $\Box$
  $\mid (CursorNotVisible \wedge c = nl) \longrightarrow CursorNotVisibleAndNewLine(prevCurX)$
  $\Box$
  $\mid (CursorNotVisible \wedge c \neq nl) \longrightarrow Pan$
  fi ;
  $SetDocCursor$; $return(\text{"OK"})$
fi

References

[1] Abrial, J-R., *B User Manual*, B.P. Project Report, Programming Research Group, Oxford University, 1988.

[2] Abrial, J-R., Gardiner, P., Morgan, C.C., Spivey, J.M., 'A Formal Approach To Large Software Construction', Programming Research Group, Oxford University, 1988.

[3] Barrett, G., *Formal Methods Applies to a Floating-Point Number System*, Technical Monograph **PRG-58**, Programming Research Group, Oxford University, 1987.

[4] Cottam, I.D., & al., 'Project Support Environments for Formal Methods', *Integrated Project Support Environments*, McDermid, J.A. (ed.), Peter Peregrinus Ltd., 1985.

[5] Dijkstra, E.G., *A Discipline of Programming*, Prentice-Hall International, 1976.

[6] Embley, D.W., Nagy, G., 'Behavioural Aspects of Text Editors', *ACM Computing Surveys 13*, 1981.

[7] Gries, D., *The Science of Programming*, Springer-Verlag, 1981.

[8] Hayes, I.J. (ed.), *Specification Case Studies*, Prentice-Hall International, 1987.

[9] Hehner, E.C.R., *The Logic of Programming*, Prentice-Hall International, 1984.

[10] He, Jifeng, 'Specification of Abstract Data Types', Seminar, Programming Research Gronp, Oxford University, 1988.

[11] Henderson, P., 'Functional Programming, Formal Specification, and Rapid Prototyping', *IEEE Trans. Soft. Eng.*, **SE-12**(2), 1986.

[12] Hoare, C.A.R., 'Proof of Correctness of Data Representations', *Acta Informatica 1*, 1972.

[13] Hoare, C.A.R., Informal Communication, M.Sc. lecture, Programming Research Group, Oxford University, 1985.

[14] Hoare, C.A.R., He, Jifeng, Sanders, J.W., 'Data Refinement Refined - Resumé', Programming Research Gronp, Oxford University, 1987.

[15] Horning, J.J., 'Putting Formal Specifications to Productive Use', *Proceedings of the Joint IBM/University of Newcastle upon Tyne Seminar*, 1983.

[16] Johnson, P., *Experience of formal development in CICS*, IBM UK Laboratories Ltd., 1988.

[17] Jones, C.B., *Systematic Software Development Using VDM*, Prentice-Hall International, 1986.

[18] Kernighan, B.W., Richie, D.M., *The C Programming Language*, Prentice-Hall Software Series, 1978.

[19] King, S., Sørensen, I.H., Woodcock, J.C.P., 'Z: Grammar and Concrete and Abstract Syntaxes', Programming Research Group, Oxford University, 1987.

[20] Morgan, C.C., 'The Schema Language', Programming Research Group, Oxford University, 1984.

[21] Morgan, C.C., 'The Specification Statement', *ACM TOPLAS 10*, 1988.

[22] Morgan, C.C., Sufrin, B.A., 'Specification of the UNIX filing system', *IEEE Trans. Soft. Eng.*, **SE-10**(2), 1984. Group, Oxford University, 1984.

[23] Morris, J., 'A Theoretical Basis for Stepwise Refinement and the Programming Calculus', *Sci. Computer Programming 9*, 1987.

[24] Naur, P., 'Intuition in software development', *Proc. Int. Joint. Conf. on Theory and Practice of Software Development (TAPSOFT)*, 1985 (LNCS 186).

[25] Neilson, D.S., *Formal Specification Of An Occam Editor*, M.Sc. Thesis, Programming Research Group, Oxford University, 1985.

[26] Partsch, H., Steinbrüggen, R., 'Program Transformation Systems', *ACM Computing Surveys 15*, 1983.

[27] Spivey, J.M., *Understanding Z: A Specification Language and its Formal Semantics*, Cambridge University Press, 1988.

[28] Spivey, J.M., *The Z Notation - A Reference Manual*, Prentice-Hall International, 1989.

[29] Sufrin, B.A., 'Formal Specification of a Display-Oriented Editor', *Science of Computer Programming 1*, 1982.

[30] Sufrin, B.A., Morgan, C.C., Sørensen, I.H., Hayes, I.J., 'Notes for a Z Handbook, Part 1 - Mathematical Language', Programming Research Group, Oxford University, 1985.

[31] Turner, D.A., 'Functional programs as executable specifications', *Mathematical Logic and Programming Languages*, C.A.R. Hoare and J.C. Shepherdson (eds.), Prentice-Hall, 1984.

[32] Wadler, P., Implementation of the VED text editor, Programming Research Group, Oxford University, 1984.

[33] Wirth, N., 'Program development by stepwise refinement', *Comm. ACM vol. 14, no. 4*, 1971.

[34] Wordsworth, J.B., *A Z Development Method*, IBM UK Laboratories Ltd., 1987.

Appendix A

Summary Of Abstract State Hierarchies

Doc1

$Pair_{Char}$
$Pair_{Word}$
$Pair_{Line}$

$Left_{Char} = \mathsf{FW}\ Left_{Word} = \mathsf{FL}\ Left_{Line}$
$Right_{Char} = \mathsf{FW}\ Right_{Word} = \mathsf{FL}\ Right_{Line}$

UD

$UDLines : seq_1\ DispLine$
$UDCurX, UDCurY : \mathbb{N}_1$
$UDCurLine : DispLine$

$UDCurY \leq \#\ UDLines$
$UDCurLine = UDLines\ UDCurY$
$UDCurX \leq \#\ UDCurLine + 1$

Doc2

Doc1
UD

$UDLines = \mathsf{FDL}^{-1}(Left_{Char} \frown Right_{Char})$
$UDCurLine = \mathsf{last}\ (\mathsf{FDL}^{-1}\ Left_{Char}) \frown \mathsf{first}\ (\mathsf{FDL}^{-1}\ Right_{Char})$
$UDCurY = \#(\mathsf{FDL}^{-1}\ Left_{Char})$
$UDCurX = \#(\mathsf{last}\ \mathsf{FDL}^{-1}\ Left_{Char}) + 1$

Doc3

Doc2

$\forall\ i : 1 \mathinner{..} \#\ UDLines - \{UDCurY\} \bullet \mathsf{visible}\ (UDLines\ i)$
$\mathsf{visible}\ (UDCurLine\ \mathsf{after}\ UDCurX - 1)$
$\mathsf{visibleseq}\ (UDLines\ \mathsf{after}\ UDCurY)$

$QP \ \widehat{=} \ [\ QPCurX, QPCurY : \mathbf{N}_1\ ]$

Doc4

$$\begin{array}{l} Doc3 \\ QP \\ \hline UDCurX = QPCurX \\ UDCurY = QPCurY \end{array}$$

MarkedText

$$\begin{array}{l} MarkSeq, MarkedSeq : seq\ Char \\ Pair_{Char} \\ \hline \quad MarkSeq = MarkedSeq = < > \\ \vee \\ \quad MarkSeq \frown MarkedSeq = Left_{Char} \\ \vee \\ \quad MarkedSeq \frown MarkSeq = Right_{Char} \end{array}$$

$Doc5 \ \widehat{=} \ Doc4 \wedge MarkedText$

$PasteBuffer \ \widehat{=} \ [\ PBuff : seq\ Char\ ]$

$Doc6 \ \widehat{=} \ Doe5 \wedge PasteBuffer$

QuoteBuffer

$$\begin{array}{l} Left_{Quote}, Right_{Quote} : seq\ Char \\ QBuff : seq\ Char \\ \hline QBuff = Left_{Quote} \frown Right_{Quote} \end{array}$$

$DocState \ \widehat{=} \ [\ State : \{State_{Doc}, State_{Quote}\}\ ]$

$DocName \ \widehat{=} \ [\ Name : seq\ Char\ ]$

$Doc7 \ \widehat{=} \ Doc6 \wedge QuoteBuffer \wedge DocState \wedge DocName$

$SearchBuffer \ \widehat{=} \ [\ SBuff : seq\ Char\ ]$

$ReplaceBuffer \ \widehat{=} \ [\ RBuff : seq\ Char\ ]$

$Doc8 \ \widehat{=} \ Doc7 \wedge SearchBuffer \wedge ReplaceBuffer$

$$WindowOffset \mathrel{\widehat=} [\, OffsetX, OffsetY : \mathbf{N} \,]$$

WindowCursor

$$WinCurX, WinCurY : \mathbf{N}_1$$

$$1 \leq WinCurX \leq WinWidth$$
$$1 \leq WinCurY \leq WinHeight$$

Window

$$WindowLines : seq\ Line$$
$$WindowOffset$$
$$WindowCursor$$

$$\# WindowLines \leq WinHeight$$
$$\forall\ y : 1 \mathinner{..} \# WinLines \bullet \#(WindowLines\ y) \leq WinWidth$$

Doc9

$$Doc8$$
$$Window$$

$$WinCurX, WinCurY = UDCurX - OffsetX, UDCurY - OffsetY$$
$$WindowLines = (WinMaskLines \text{ after } OffsetY) \text{ for } WinHeight$$

where

$$\# WinMaskLines = \# UDLines$$
$$\forall\ y : 1 \mathinner{..} \# UDLines \bullet$$
$$WinMaskLines\ y = ((UDLines\ y) \text{ after } OffsetX) \text{ for } WinWidth$$

Appendix B

Summary Of Concrete State Hierarchies

$$
\begin{array}{|l}
\textit{ConcDoc1} \\
\hline
Arr : 1 \,..\, Max \rightarrow Char \\
LP, RP, CP : 0 \,..\, Max \\
\hline
LP \leq RP \\
CP \leq Max + LP - RP \\
\hline
\end{array}
$$

$$ArrCont \mathrel{\widehat=} (Arr \text{ for } LP) \frown (Arr \text{ after } RP)$$

Rel_{Doc1}:

$$
\left[\begin{array}{lcl}
Left_{Char} & \widehat= & ArrCont \text{ for } CP \\
Right_{Char} & \widehat= & ArrCont \text{ after } CP
\end{array}\right.
$$

$Rel_{Doc1\,Standard}$:

$$
\left[\begin{array}{lcl}
Left_{Char} & \widehat= & Arr \text{ for } LP \\
Right_{Char} & \widehat= & Arr \text{ after } RP
\end{array}\right.
$$

$$
\begin{array}{|l}
\textit{ConcDoc2} \\
\hline
ConcDoc1 \\
StartIn, EndIn, DocNL, WSRem, NLRem : 0 \,..\, Max \\
CurX, CurY : 1 \,..\, Max + 1 \\
\hline
StartIn \leq CP \leq EndIn \\
\mathsf{NoNLin}\ (ArrCont, StartIn + 1 \,..\, EndIn) \\
StartIn \neq 0 \Rightarrow ArrCont\ StartIn = nl \\
EndIn \neq (Max + LP - RP) \Rightarrow ArrCont(EndIn + 1) = nl \\
CurX = CP - StartIn + 1 \\
CurY = \mathsf{NumNLin}\ (ArrCont, 1 \,..\, CP) + 1 \\
DocNL = \mathsf{TotalNLin}\ ArrCont \\
\hline
\end{array}
$$

$$\left[\begin{array}{l} Rel_{Doc1} \\ UDCurX, UDCurY = CurX, CurY \\ UDLines = \mathsf{FDL}^{-1}\ ArrCont \\ UDCurLine = StartIn + 1 \mathinner{..} EndIn \upharpoonleft ArrCont \\ \#\ UDLines = DocNL + 1 \end{array}\right.$$

ConcDoc3

$$\begin{array}{|l} ConcDoc2 \\ \hline \forall\ i : 1 \mathinner{..} DocNL + 1 - \{CurY\} \bullet \mathsf{visible}\ ((\mathsf{FDL}^{-1}\ ArrCont)\ i) \\ \mathsf{visible}\ (CP + 1 \mathinner{..} EndIn \upharpoonleft ArrCont) \\ \mathsf{visibleseq}\ (\mathsf{FDL}^{-1}(ArrCont\ \mathsf{after}\ EndIn)) \end{array}$$

Rel_{Doc3}:

$$Rel_{Doc3} \ \widehat{=}\ Rel_{Doc2}$$

ConcDoc4

$$\begin{array}{|l} ConcDoc3 \\ WSIns, NLIns : 0 \mathinner{..} Max \end{array}$$

Rel_{Doc4}:

$$\left[\begin{array}{l} Rel_{Doc3} \\ QPCurX, QPCurY = CurX, CurY \end{array}\right.$$

$ConcDoc6$

$$
\begin{array}{l}
ConcDoc4 \\
PArr : 1 \,..\, Max \;\rightarrowtail\; Char \\
PP : 0 \,..\, Max \\
MP : -1 \,..\, Max
\end{array}
$$

Rel_{Doc6}:

$$
\left[
\begin{array}{l}
Rel_{Doc4} \\
MP = -1 \;\Rightarrow\; MarkSeq = MarkedSeq = \langle\,\rangle \\
MP \neq -1 \;\wedge\; MP \leq CP \;\Rightarrow\; MarkSeq = ArrCont \text{ for } MP \\
\qquad\qquad\qquad\qquad\qquad\quad MarkedSeq = MP + 1 \,..\, CP \upharpoonleft ArrCont \\
MP \neq -1 \;\wedge\; MP > CP \;\Rightarrow\; MarkSeq = ArrCont \text{ after } MP \\
\qquad\qquad\qquad\qquad\qquad\quad MarkedSeq = CP + 1 \,..\, MP \upharpoonleft ArrCont \\
PBuff = PArr \text{ for } PP
\end{array}
\right.
$$

$ConcDoc8$

$$
\begin{array}{l}
ConcDoc6 \\
QArr, SArr, RpArr : 1 \,..\, QMax \;\rightarrowtail\; Char \\
QP, QCP, SP, RpP : 0 \,..\, QMax \\
FName : String \\
MatchedLength : 0 \,..\, QMax \\
EState : \{\, State_{Doc}, State_{Quote} \,\} \\
DocChanged : \mathbf{B} \\
\hline
QCP \leq QP \\
(SArr \text{ for } SP) \text{ matches } (ArrCont \text{ after } CP) \;\Rightarrow \\
\qquad MatchedLength = \text{matchedlength } (SArr \text{ for } SP, ArrCont \text{ after } CP)
\end{array}
$$

Rel_{Doc8}:

$$
\left[
\begin{array}{l}
Rel_{Doc6} \\
State = EState \\
Left_{Quote} = QArr \text{ for } QCP \\
Right_{Quote} = QCP + 1 \,..\, QP \upharpoonleft QArr \\
SBuff = SArr \text{ for } SP \\
RBuff = RpArr \text{ for } RpP \\
QBuff = QArr \text{ for } QP \\
FileName = FName
\end{array}
\right.
$$

$TerminalCursor \ \widehat{=} \ [\ TermCurX, TermCurY : \mathbb{N}\ ]$

$$
\begin{array}{|l}
\multicolumn{1}{l}{TerminalWindow} \\
\hline
TermWinLines : 1 \,..\, WinHeight \rightarrow (1 \,..\, WinWidth \rightarrow Char) \\
\hline
\end{array}
$$

$TerminalDisplay \ \widehat{=} \ TerminalCursor \wedge TerminalWindow$

$ConcWinOffset \ \widehat{=} \ [\ WinOffX, WinOffY : 0 \,..\, Max\ ]$

$$
\begin{array}{|l}
\multicolumn{1}{l}{ConcDoc9} \\
\hline
ConcDoc8 \\
TerminalDisplay \\
ConcWinOffset \\
WinStartln : 0 \,..\, Max \\
WinLineOK : 1 \,..\, WinHeight \rightarrow \mathbb{B} \\
\hline
\mathsf{NumNLin}\ (ArrCont, 1 \,..\, WinStartln) = WinOffY \\
WinStartln \neq 0 \ \Rightarrow \ ArrCont\ WinStartln = nl \\
TermCurX = CurX - WinOffX \in 1 \,..\, WinWidth \\
TermCurY = CurY - WinOffY \in 1 \,..\, WinHeight \\
\forall\ y : 1 \,..\, WinHeight \bullet \\
\quad WinLineOK\ y \ \Rightarrow \ \mathsf{Displayed}\ (ArrCont, y, WinOffX, WinOffY) \\
\hline
\end{array}
$$

Rel_{Doc9}:

$$
\left[
\begin{array}{l}
Rel_{Doc8} \\
OffsetX, OffsetY = WinOffX, WinOffY \\
WinCurX, WinCurY = CurX - WinOffX, CurY - WinOffY \\
WindowLines = TermWinLines
\end{array}
\right.
$$

Appendix C

Implementation Of The Editor Specification

```
                    /* IMPLEMENTATION OF DOC1 */
                  /* ========================== */

#include "/se/daven/ox/c/c2/externglobals.c"

/* ------------------------------------------------------------------------- */
int OP_Doc1()
{
   if      (OP==LeftMoveChar)      { OPType=LeftMove; return(LeftMoveChar_Doc1()); }
   else if (OP==RightMoveChar)     { OPType=RightMove; return(RightMoveChar_Doc1()); }
   else if (OP==LeftMoveWord)      { OPType=LeftMove; return(LeftMoveWord_Doc1()); }
   else if (OP==RightMoveWord)     { OPType=RightMove; return(RightMoveWord_Doc1()); }
   else if (OP==LeftMoveLine)      { OPType=LeftMove; return(LeftMoveLine_Doc1()); }
   else if (OP==RightMoveLine)     { OPType=RightMove; return(RightMoveLine_Doc1()); }
   else if (OP==LeftDeleteChar)    { OPType=LeftDelete; return(LeftDeleteChar_Doc1()); }
   else if (OP==RightDeleteChar)   { OPType=RightDelete; return(RightDeleteChar_Doc1()); }
   else if (OP==LeftDeleteWord)    { OPType=LeftDelete; return(LeftDeleteWord_Doc1()); }
   else if (OP==RightDeleteWord)   { OPType=RightDelete; return(RightDeleteWord_Doc1()); }
   else if (OP==LeftDeleteLine)    { OPType=LeftDelete; return(LeftDeleteLine_Doc1()); }
   else if (OP==RightDeleteLine)   { OPType=RightDelete; return(RightDeleteLine_Doc1()); }
   else if (OP==InsertChar)        { OPType=LeftInsert; return(InsertChar_Doc1()); }
   else if (OP==MoveToTop)         { OPType=LeftMove; return(MoveToTop_Doc1()); }
   else if (OP==MoveToBot)         { OPType=RightMove; return(MoveToBot_Doc1()); }
}
/* ------------------------------------------------------------------------- */
int Initialize_Doc1()
{
 LP=0; CP=0; RP=Max;
}
/* ------------------------------------------------------------------------- */
int GetArrCont(ptr)   int ptr;
{
 if   (ptr<=LP) { return(Arr[ptr]); }
 else           { return(Arr[ptr+RP-LP]); }
}
/* ------------------------------------------------------------------------- */
Standardize_Doc1()
{
 if      (LP>CP) { while (LP>CP) { Arr[RP]=Arr[LP]; RP--; LP--; } }
 else if (LP<CP) { while (LP<CP) { Arr[LP+1]=Arr[RP+1]; RP++; LP++; } }
}
/* ------------------------------------------------------------------------- */
int RightMoveChar_Doc1()
{
 if (CP!=Max+LP-RP) { CP++; return(OK); }
 else               { return(Bot); }
}
/* ------------------------------------------------------------------------- */
int LeftMoveChar_Doc1()
{
 if (CP!=0) { CP--; return(OK); }
 else       { return(Top); }
}
/* ------------------------------------------------------------------------- */
```

```
int RightMoveWord_Doc1()
{
 if (CP!=Max+LP-RP) { if      (GetArrCont(CP+1)==nl) { CP++; }
                      else if (GetArrCont(CP+1)==sp) { RMWSWord(); }
                      else                            { RMNWSWord(); }
                      return(OK);
                    }
 else               { return(Bot); }
}
/* ------------------------------------------------------------------------------- */
RMWSWord()
{
 while (CP<Max+LP-RP && GetArrCont(CP+1)==sp) { CP++; }
}
/* ------------------------------------------------------------------------------- */
RMNWSWord()
{
 while (CP<Max+LP-RP && GetArrCont(CP+1)!=sp && GetArrCont(CP+1)!=nl) { CP++; }
}
/* ------------------------------------------------------------------------------- */
int LeftMoveWord_Doc1()
{
 if (CP!=0) { if      (GetArrCont(CP)==nl) { CP--; }
              else if (GetArrCont(CP)==sp) { LMWSWord(); }
              else                         { LMNWSWord(); }
              return(OK);
            }
 else       { return(Top); }
}
/* ------------------------------------------------------------------------------- */
LMWSWord()
{
 while (CP>0 && GetArrCont(CP)==sp) { CP--; }
}
/* ------------------------------------------------------------------------------- */
LMNWSWord()
{
 while (CP>0 && GetArrCont(CP)!=sp && GetArrCont(CP)!=nl) { CP--; }
}
/* ------------------------------------------------------------------------------- */
int RightMoveLine_Doc1()
{
 if   (CP!=Max+LP-RP) { if (GetArrCont(CP+1)==nl)
                                CP++;
                        else
                                while (CP!=Max+LP-RP && GetArrCont(CP+1)!=nl) { CP++; }
                        return(OK);
                      }
 else                 { return(Bot); }
}
/* ------------------------------------------------------------------------------- */
int LeftMoveLine_Doc1()
{
 if (CP!=0) { if (GetArrCont(CP)==nl) CP--;
              else                    while (CP!=0 && GetArrCont(CP)!=nl) { CP--; }
              return(OK);
            }
 else       { return(Top); }
}
/* ------------------------------------------------------------------------------- */
int RightDeleteChar_Doc1()
{
 Standardize_Doc1(); if   (RP!=Max) { RP++; return(OK); }
                     else           { return(Bot); }
```

```
}
/* ------------------------------------------------------------------------------ */
int LeftDeleteChar_Doc1()
{
 Standardize_Doc1(); if   (LP!=0) { LP--; CP--; return(OK); }
                    else          { return(Top); }
}
/* ------------------------------------------------------------------------------ */
int RightDeleteWord_Doc1()
{
 Standardize_Doc1();
 if (RP!=Max) { if       (Arr[RP+1]==nl) RP++;
                else if (Arr[RP+1]==sp) while (RP!=Max && Arr[RP+1]==sp) { RP++; }
                else                    while (RP!=Max && Arr[RP+1]!=nl && Arr[RP+1]!=sp)
                                                                        { RP++; }
                return(OK);
              }
 else         { return(Bot); }
}
/* ------------------------------------------------------------------------------ */
int LeftDeleteWord_Doc1()
{
 Standardize_Doc1();
 if (LP!=0) { if       (Arr[CP]==nl) CP--;
              else if (Arr[CP]==sp) while (CP!=0 && Arr[CP]==sp) { CP--; }
              else                  while (CP!=0 && Arr[CP]!=nl && Arr[CP]!=sp) { CP--; }
              LP=CP; return(OK);
            }
 else       { return(Top); }
}
/* ------------------------------------------------------------------------------ */
int RightDeleteLine_Doc1()
{
 Standardize_Doc1();
 if (RP!=Max) { if   (Arr[RP+1]==nl) RP++;
                else                  while (RP!=Max && Arr[RP+1]!=nl) { RP++; }
                return(OK);
              }
 else         { return(Bot); }
}
/* ------------------------------------------------------------------------------ */
int LeftDeleteLine_Doc1()
{
  Standardize_Doc1();
 if (LP!=0) { if   (Arr[LP]==nl) LP--;
              else                while (LP!=0 && Arr[LP]!=nl) { LP--; }
              CP=LP; return(OK);
            }
 else       { return(Top); }
}
/* ------------------------------------------------------------------------------ */
int InsertChar_Doc1()
{
 int ptr=CP;   int count=0;
 Standardize_Doc1();
 if      (LP!=RP && OPChar!=TAB) { LP++; CP++; Arr[LP]=OPChar; return(OK); }
 else if (LP!=RP && OPChar==TAB) { while (ptr!=0 && Arr[ptr]!=nl) { count++; ptr--; }
                                   count=tabstop-(count%tabstop);
                                   while (count!=0 && LP!=RP) { Arr[LP+1]=sp;
                                                                CP++; LP++; count--;
                                                              }
                                   return(OK);
                                 }
 else                            { return(Full); }
```

```
}
/* ---------------------------------------------------------------------------------- */
int MoveToTop_Doc1()
{
 if   (CP!=0) { CP=0; return(OK); }
 else         { return(Top); }
}
/* ---------------------------------------------------------------------------------- */
int MoveToBot_Doc1()
{
 if   (CP!=Max+LP-RP) { CP=(Max+LP-RP); return(OK); }
 else                 { return(Bot); }
}
/* ---------------------------------------------------------------------------------- */
```

```
                              /* IMPLEMENTATION OF DOC3 */
                            /* ========================= */

#include "/se/daven/ox/c/c2/externglobals.c"

/* ---------------------------------------------------------------------------------- */
int OP_Doc3()
{
 int prevCP=CP;   int prevLP=LP;   int prevRP=RP;   int rep=OP_Doc1();
 Update_Doc3(prevCP,prevLP,prevRP); return(rep);
}
/* ---------------------------------------------------------------------------------- */
int Initialize_Doc3()
{
 Initialize_Doc1(); Startln=0; Endln=0; CurX=1; CurY=1; DocNL=0; WSRem=0; NLRem=0;
}
/* ---------------------------------------------------------------------------------- */
SetStartln()
{
 Startln=CP; while (Startln>0 && GetArrCont(Startln)!=nl) { Startln--; }
}
/* ---------------------------------------------------------------------------------- */
SetEndln()
{
 Endln=CP; while (Endln<Max+LP-RP && GetArrCont(Endln+1)!=nl) { Endln++; }
}
/* ---------------------------------------------------------------------------------- */
Update_Doc3(prevCP,prevLP,prevRP)   int prevCP;   int prevLP;   int prevRP;
{
 int NumNL;   WSRem=0; NLRem=0;
 if (OPType!=NoMove)
       { SetStartln(); SetEndln(); CurX=CP-Startln+1;
         if      (OPType==LeftInsert)  { NumNL=NLCountArr(prevCP,CP); CurY=CurY+NumNL;
                                         DocNL=DocNL+NumNL;
                                       }
         else if (OPType==RightInsert) { DocNL=DocNL+NLCountArr(RP,LP-prevLP+prevRP); }
         else if (OPType==LeftDelete)  { NumNL=NLCountArr(CP,prevCP);  DocNL=DocNL-NumNL;
                                         CurY=CurY-NumNL;
                                       }
         else if (OPType==RightDelete) { Standardize_Doc1();
                                         DocNL=DocNL-NLCountArr(LP-prevLP+prevRP,RP);
                                       }
```

```
        else if (OPType==LeftMove)    { if (CP==0) { CurY=1; }
                                        else       { CurY=CurY-NLCountArrCont(CP,prevCP);}
                                      }
        else if (OPType==RightMove)   { if (CP==Max+LP-RP)
                                              { CurY=DocNL+1; }
                                        else
                                              { CurY=CurY+NLCountArrCont(prevCP,CP); }
                                      }
        RemTrailWS(prevCP); RemTrailNL();
      }
}
/* ------------------------------------------------------------------------------- */
int NLCountArr(first,last)   register int first;   int last;
{
 int NumNL=0;
 while (first!=last) { first++; if (Arr[first]==nl) { NumNL++; } }
 return(NumNL);
}
/* ------------------------------------------------------------------------------- */
int NLCountArrCont(first,last)   register int first;   int last;
{
 int NumNL=0;
 while (first!=last) { first++; if (GetArrCont(first)==nl) { NumNL++; } }
 return(NumNL);
}
/* ------------------------------------------------------------------------------- */
RemTrailWS(prevCP)   int prevCP;
{
 int tempCP=CP;   int tempEndln=Endln;   int prevEndln;
 CP=prevCP; SetEndln(); prevEndln=Endln; CP=tempCP; Endln=tempEndln;
 if (OPType==LeftMove || OPType==RightMove || OPType==LeftInsert || OPType==RightInsert)
      { if (prevEndln==Endln)
             { if (Endln!=CP && GetArrCont(Endln)==sp)
                     { CP=Endln; Standardize_Doc1(); CP=tempCP;
                       while (Endln!=CP && Arr[Endln]==sp) { Endln--; LP--; WSRem++; }
                     }
             }
        else
             { if (prevEndln!=0 && GetArrCont(prevEndln)==sp)
                     { CP=prevEndln; Standardize_Doc1(); CP=tempCP;
                       while (LP!=0 && Arr[LP]==sp) { LP--; WSRem++; }
                       if (CP>LP) { CP=CP-WSRem; Startln=Startln-WSRem;
                                    Endln=Endln-WSRem;
                                  }
                     }
             }
      }
}
/* ------------------------------------------------------------------------------- */
RemTrailNL()
{
 int tempCP=CP;
 if (OPType==LeftMove || OPType==RightInsert)
      { if (CP!=Max+LP-RP && GetArrCont(Max+LP-RP)==nl)
              { CP=Max+LP-RP; Standardize_Doc1(); CP=tempCP;
                while (LP!=CP && Arr[LP]==nl)  { LP--; DocNL--; NLRem++;}
              }
      }
}
/* ------------------------------------------------------------------------------- */
```

```
                         /* IMPLEMENTATION OF DOC4 */
                       /* ========================= */

#include "/se/daven/ox/c/c2/externglobals.c"

/* ---------------------------------------------------------------------------- */
int OP_Doc4()
{
 int rep;
 if      (OP==CursorUpLine)    { rep=CursorUpLine_Doc4(); OPType=LeftMove; }
 else if (OP==CursorUpPage)    { rep=CursorUpPage_Doc4(); OPType=LeftMove; }
 else if (OP==CursorDownLine)  { rep=CursorDownLine_Doc4(); OPType=RightMove; }
 else if (OP==CursorDownPage)  { rep=CursorDownPage_Doc4(); OPType=RightMove; }
 else if (OP==CursorLeftChar)  { rep=CursorLeftChar_Doc4(); OPType=LeftMove; }
 else if (OP==CursorRightChar) { rep=CursorRightChar_Doc4(); OPType=RightMove; }
 else                          { rep=OP_Doc3(); Update_Doc4(); }
 return(rep);
}
/* ---------------------------------------------------------------------------- */
int Initialize_Doc4()
{
 Initialize_Doc3(); WSIns=0; NLIns=0;
}
/* ---------------------------------------------------------------------------- */
Update_Doc4()
{
 WSIns=0; NLIns=0;
}
/* ---------------------------------------------------------------------------- */
int CursorUpLine_Doc4()
{
 return(CursorUp(1));
}
/* ---------------------------------------------------------------------------- */
int CursorUpPage_Doc4()
{
 return(CursorUp(PageHeight));
}
/* ---------------------------------------------------------------------------- */
int CursorUp(y)   int y;
{
 int prevCP=CP;   int prevLP=LP;   int prevRP=RP;   int prevCurX=CurX;   int prevCurY=CurY;
 int prevStartln=Startln;   int prevEndln=Endln;   int prevOP=OP;      int cumWSRem=0;
 int cumNLRem=0;            int rep;
 WSIns=0; NLIns=0;
 if (CurY!=1)
      { OP=LeftMoveLine;
        while (CurY!=prevCurY-y && CP!=0) { OP_Doc3(); UpdatecumWSNL(&cumWSRem,&cumNLRem);}
        if      (prevCurX<=Endln-Startln+1)
                    { CurX=prevCurX; CP=Startln+CurX-1; rep=OK; }
        else if (prevCurX>Endln-Startln+1 && LP+prevCurX-1-Endln+Startln<=RP)
                    { CP=Endln; CurX=Endln-Startln+1; OP=InsertChar; OPChar=sp;
                      while (CurX!=prevCurX) { OP_Doc3(); WSIns++;
                                               UpdatecumWSNL(&cumWSRem,&cumNLRem);
                                          }
                      rep=OK;
                    }
        else
                    { CP=prevCP; LP=prevLP; RP=prevRP; CurX=prevCurX; CurY=prevCurY;
                      Startln=prevStartln; Endln=prevEndln; rep=Full;
                    }
```

```
        WSRem=cumWSRem; NLRem=cumNLRem; OP=prevOP; return(rep);
      }
 else
      { Update_Doc3(prevCP,prevLP,prevRP); Update_Doc4(); return(TopLine); }
}
/* ------------------------------------------------------------------------------- */
int CursorDownLine_Doc4()
{
 return(CursorDown(1));
}
/* ------------------------------------------------------------------------------- */
int CursorDownPage_Doc4()
{
 return(CursorDown(PageHeight));
}
/* ------------------------------------------------------------------------------- */
int CursorDown(y)   int y;
{
 int prevCP=CP;   int prevLP=LP;   int prevRP=RP;   int prevCurX=CurX;   int prevCurY=CurY;
 int prevStartln=Startln;   int prevEndln=Endln;   int prevOP=OP;     int cumWSRem=0;
 int cumNLRem=0;                int rep;
 WSIns=0; NLIns=0;  OP=RightMoveLine;
 while (CurY!=prevCurY+y && CP!=Max+LP-RP)
            { OP_Doc3(); UpdatecumWSNL(&cumWSRem,&cumNLRem);}
 if (CurY==prevCurY+y && LP+prevCurX-Endln+Startln-1<=RP)
            { if (prevCurX<=Endln-Startln+1)
                        { CurX=prevCurX; CP=Startln+CurX-1; }
              else
                        { CurX=Endln-Startln+1; CP=Endln; OP=InsertChar; OPChar=sp;
                          while (CurX!=prevCurX) { OP_Doc3(); WSIns++;
                                                   UpdatecumWSNL(&cumWSRem,&cumNLRem);
                                                 }
                        }
              rep=OK;
            }
 else if (CurY!=prevCurY+y && LP+prevCurY+y-DocNL-1+prevCurX-1<=RP)
            { OP=InsertChar; OPChar=nl;
              while (CurY!=prevCurY+y) { OP_Doc3(); NLIns++;
                                         UpdatecumWSNL(&cumWSRem,&cumNLRem);
                                       }
              OPChar=sp; while (CurX!=prevCurX) { OP_Doc3(); WSIns++;
                                                  UpdatecumWSNL(&cumWSRem,&cumNLRem);
                                                }
              rep=OK;
            }
 else
            { CP=prevCP; LP=prevLP; RP=prevRP; CurX=prevCurX; CurY=prevCurY;
              Startln=prevStartln; Endln=prevEndln; rep=Full;
            }
 WSRem=cumWSRem; OP=prevOP; return(rep);

}
/* ------------------------------------------------------------------------------- */
int CursorLeftChar_Doc4()
{
 int rep;
 WSIns=0; NLIns=0; OP=LeftMoveChar; rep=OP_Doc3(); OP=CursorLeftChar; return(rep);
}
/* ------------------------------------------------------------------------------- */
int CursorRightChar_Doc4()
{
 int rep;
 WSIns=0; NLIns=0;
 if   (CP!=Endln) { OP=RightMoveChar; rep=OP_Doc3(); }
```

```
 else              { OP=InsertChar; OPChar=sp; rep=OP_Doc3(); WSIns++; }
 OP=CursorRightChar; return(rep);
}
/* ----------------------------------------------------------------------------------- */
UpdatecumWSNL(cumWSRem,cumNLRem)  int *cumWSRem;  int *cumNLRem;
{
 *cumWSRem=(*cumWSRem)+WSRem; *cumNLRem=(*cumNLRem)+NLRem;
}
/* ----------------------------------------------------------------------------------- */
```

```
                         /* IMPLEMENTATION OF DOC6 */
                       /* ====================== */

#include "/se/daven/ox/c/c2/externglobals.c"

/* ----------------------------------------------------------------------------------- */
int OP_Doc6()
{
 int prevCP=CP;   int prevLP=LP;   int prevRP=RP;  int rep;
 if      (OP==Mark)  { OPType=NoMove; rep=Mark_Doc6(); Update_Doc3(prevCP,prevLP,prevRP);
                        Update_Doc4();
                     }
 else if (OP==Lift)  { OPType=NoMove; rep=Lift_Doc6(); Update_Doc3(prevCP,prevLP,prevRP);
                        Update_Doc4(); }
 else if (OP==Cut)   { if (CP>MP) { OPType=LeftDelete; } else { OPType=RightDelete; }
                        rep=Cut_Doc6(); Update_Doc3(prevCP,prevLP,prevRP); Update_Doc4();
                     }
 else if (OP==Paste) { OPType=LeftInsert;
                       rep=Paste_Doc6(); Update_Doc3(prevCP,prevLP,prevRP); Update_Doc4();
                     }
 else                { rep=OP_Doc4(); Update_Doc6(prevCP); }
 return(rep);
}
/* ----------------------------------------------------------------------------------- */
int Initialize_Doc6()
{
 Initialize_Doc4(); PP=0; MP=(-1);
}
/* ----------------------------------------------------------------------------------- */
Update_Doc6(prevCP)   int prevCP;
{
 if ((OPType!=NoMove && OPType!=LeftMove && OPType!=RightMove) || MP>Max+LP-RP)
      { MP=(-1); }
 else
      { if      (prevCP<=MP && CP<=MP) { MP=MP-WSRem+WSIns; }
        else if (prevCP<=MP && CP>=MP) { MP=MP-WSRem; }
        else if (prevCP>=MP && CP<=MP) { MP=MP+WSIns; }
      }
}
/* ----------------------------------------------------------------------------------- */
int Mark_Doc6()
{
 MP=CP; return(OK);
}
/* ----------------------------------------------------------------------------------- */
int Lift_Doc6()
{
```

```
 if   (CP!=MP && MP!=(-1)) { CopyMTextPBuff(); return(OK); }
 else                      { return(NoTextMarked); }
}
/* ------------------------------------------------------------------------------- */
CopyMTextPBuff()
{
 int MPptr; PP=0;
 if      (CP<MP) { MPptr=CP;
                   while (MPptr!=MP) { MPptr++; PP++; PArr[PP]=GetArrCont(MPptr); }
                 }
 else if (CP>MP) { MPptr=MP;
                   while (MPptr!=CP) { MPptr++; PP++; PArr[PP]=GetArrCont(MPptr); }
                 }
}
/* ------------------------------------------------------------------------------- */
int Cut_Doc6()
{
 if (MP!=(-1) && MP!=CP) { CopyMTextPBuff(); RemMText(); return(OK); }
 else                    { return(NoTextMarked); }
}
/* ------------------------------------------------------------------------------- */
RemMText()
{
 Standardize_Doc1(); if      (MP<CP) { LP=MP; CP=MP; }
                     else if (MP>CP) { RP=RP+MP-LP; }
 MP=(-1);
}
/* ------------------------------------------------------------------------------- */
int Paste_Doc6()
{
 int PPptr;
 if      (PP==0)
           { return(PBuffEmpty); }
 else if (PP!=0 && PP>RP-LP)
           { return(Full); }
 else
           { Standardize_Doc1(); MP=CP; PPptr=0;
             while (PPptr!=PP) { PPptr++; LP++; CP++; Arr[LP]=PArr[PPptr]; }
             return(OK);
           }
}
/* ------------------------------------------------------------------------------- */
```

```
                           /* IMPLEMENTATION OF DOC8 */
                           /* ====================== */

#include "/se/daven/ox/c/c2/externglobals.c"
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>

/* ------------------------------------------------------------------------------- */
int OP_Doc8()
{
 int prevCP=CP;  int prevLP=LP;  int prevRP=RP;  int rep;
 if (EState==State_Doc)
     { if      (OP==DownSearch) { OPType=RightMove; rep=DownSearch_Doc8();
```

```
                              Update_Doc3(prevCP,prevLP,prevRP);
                               Update_Doc4(); Update_Doc6(prevCP);
                             }
        else if (OP==UpSearch)  { OPType=LeftMove; rep=UpSearch_Doc8();
                              Update_Doc3(prevCP,prevLP,prevRP);
                              Update_Doc4(); Update_Doc6(prevCP);
                             }
        else if (OP==Replace)   { OPType=RightDelete;
                              rep=Replace_Doc8(&prevCP,&prevLP,&prevRP);
                              OPType=LeftInsert; Update_Doc3(prevCP,prevLP,prevRP);
                              Update_Doc4(); Update_Doc6(prevCP);
                             }
        else if (OP==Quote)     { rep=Quote_StateDoc(); Update_Doc3(prevCP,prevLP,prevRP);
                              Update_Doc4(); Update_Doc6(prevCP);
                             }
        else                    { rep=OP_Doc6(); Update_Doc8(); }
        return(rep);
      }
 else
      { if      (OP==InsertChar)       { rep=InsertChar_Quote(); }
        else if (OP==LeftMoveChar)     { rep=LeftMoveChar_Quote(); }
        else if (OP==CursorLeftChar)   { rep=LeftMoveChar_Quote(); }
        else if (OP==RightMoveChar)    { rep=RightMoveChar_Quote(); }
        else if (OP==CursorRightChar)  { rep=RightMoveChar_Quote(); }
        else if (OP==LeftDeleteChar)   { rep=LeftDeleteChar_Quote(); }
        else if (OP==RightDeleteChar)  { rep=RightDeleteChar_Quote(); }
        else if (OP==DownSearch)       { OPType=RightMove; rep=DownSearch_Doc8();
                                         Update_Doc3(prevCP,prevLP,prevRP);
                                         Update_Doc4(); Update_Doc6(prevCP);
                                       }
        else if (OP==UpSearch)         { OPType=LeftMove; rep=UpSearch_Doc8();
                                         Update_Doc3(prevCP,prevLP,prevRP);
                                         Update_Doc4(); Update_Doc6(prevCP);
                                       }
        else if (OP==Replace)          { OP=RightDelete;
                                         rep=Replace_Doc8(&prevCP,&prevLP,&prevRP);
                                         OPType=LeftInsert;
                                         Update_Doc3(prevCP,prevLP,prevRP);
                                         Update_Doc4(); Update_Doc6(prevCP);
                                       }
        else if (OP==Quote)            { rep=Quote_StateQuote();
                                         Update_Doc3(prevCP,prevLP,prevRP);
                                         Update_Doc4(); Update_Doc6(prevCP);
                                       }
        else                           { Update_Doc3(prevCP,prevLP,prevRP); Update_Doc4();
                                         Update_Doc6(prevCP); rep=IllegalEditOp;
                                       }
        return(rep);
      }
}
/* ------------------------------------------------------------------------------- */
int Initialize_Doc8()
{
 Initialize_Doc6(); SP=0; MatchedLength=0; RpP=0; EState=State_Doc; DocChanged=FALSE;
}
/* ------------------------------------------------------------------------------- */
Update_Doc8()
{
 if (OPType!=NoMove && OPType!=LeftMove && OPType!=RightMove)
       { if (DocChanged==FALSE) { PromptDisplayed=FALSE; }
         DocChanged=TRUE;
       }
}
/* ------------------------------------------------------------------------------- */
```

```
int DownSearch_Doc8()
{
 int prevCP=CP;   int matched=FALSE;   char schstring[QMax];
 if (EState==State_Quote) { CopyQBuffSBuff(); EState=State_Doc; }
 CopySBuffToString(schstring);
 if    (SP!=0) { SetPromptMsg(schstring); PromptMessage(SearchingDownFor);
                 while (CP<Max+LP-RP && Not(matched) && NoInterrupt)
                         { CP++; CheckForMatch(&matched); }
                 if (matched) { PromptMessage(Found); return(OK); }
                 else         { CP=prevCP; SetPromptMsg(schstring); return(NotFound); }
               }
 else          { return(SBuffEmpty); }
}
/* ------------------------------------------------------------------------------- */
int UpSearch_Doc8()
{
 int matched=FALSE;   int prevCP=CP;   char schstring[QMax];
 if (EState==State_Quote) { CopyQBuffSBuff(); EState=State_Doc; }
 CopySBuffToString(schstring);
 if    (SP!=0) { SetPromptMsg(schstring); PromptMessage(SearchingUpFor);
                 while (CP>0 && Not(matched) && NoInterrupt)
                         { CP--; CheckForMatch(&matched); }
                 if (matched) { PromptMessage(Found); return(OK); }
                 else         { CP=prevCP; SetPromptMsg(schstring); return(NotFound); }
               }
 else          { return(SBuffEmpty); }
}
/* ------------------------------------------------------------------------------- */
CheckForMatch(matched)   int *matched;
{
 int SPptr=0;   int Docptr=0; int lastmatchOK=TRUE;
 while (SPptr!=SP && CP+Docptr<Max+LP-RP && lastmatchOK)
         { lastmatchOK=CharMatched(GetArrCont(CP+Docptr+1),&SPptr); SPptr++; Docptr++; }
 if (SPptr==SP && lastmatchOK) { *matched=TRUE; MatchedLength=Docptr; }
}
/* ------------------------------------------------------------------------------- */
int CharMatched(c,ptr)   char c;   int *ptr;
{
 int prevptr=(*ptr);   int between=FALSE;   int Notflag=FALSE;   int min;   int max;
 if      (SArr[(*ptr)+1]=='.')  { return(TRUE); }
 else if (SArr[(*ptr)+1]=='\\') { (*ptr)++; return((*ptr)!=SP && SArr[(*ptr)+1]==c); }
 else if (SArr[(*ptr)+1]=='^')  { (*ptr)++; return((*ptr)!=SP && SArr[(*ptr)+1]!=c); }
 else if (SArr[(*ptr)+1]=='[')  { while ((*ptr)!=SP && SArr[(*ptr)+1]!=']') { (*ptr)++; }
                                  if ((*ptr)==SP) { return(FALSE); }
                                  if (SArr[prevptr+2]=='^') { Notflag=TRUE; prevptr++; }
                                  if ((*ptr)-prevptr==4 && SArr[(*ptr)-1]=='-')
                                       { min=SArr[(*ptr)-2]; max=SArr[*ptr];
                                         between=(c>=min && c<=max);
                                       }
                                  else
                                       {
                                         while (prevptr+1!=(*ptr) && Not(between))
                                            { prevptr++; between=(c==SArr[prevptr+1]); }
                                       }
                                  if   (Notflag) { return(Not(between)); }
                                  else           { return(between); }
                                }
 else                           { return(SArr[(*ptr)+1]==c); }
}
/* ------------------------------------------------------------------------------- */
int Replace_Doc8(prevCP,prevLP,prevRP)   int *prevCP;   int *prevLP;   int *prevRP;
{
 int ptr=0;   int matched;   char schstring[QMax];   char rplstring[QMax];
 if (EState==State_Quote) { CopyQBuffRpBuff(); EState=State_Doc; }
```

```
 CheckForMatch(&matched);
 if      (matched && RpP-MatchedLength<=RP-LP)
           { Standardize_Doc1(); CopySBuffToString(schstring);
             CopyRBuffToString(rplstring); SetPromptMsg(schstring);
             PromptMessage(Replaced); SetPromptMsg(rplstring); PromptMessage(With);
             RP=RP+MatchedLength; Update_Doc3(*prevCP,*prevLP,*prevRP); Update_Doc4();
             Update_Doc6(*prevCP,*prevLP,*prevRP);
             *prevCP=CP; *prevLP=LP; *prevRP=RP;
             while (ptr!=RpP) { LP++; ptr++; Arr[LP]=RpArr[ptr]; }
             CP=LP; DocChanged=TRUE; return(OK);
           }
 else if (matched && RpP-SP>RP-LP)
           { return(Full); }
 else
           { CopySBuffToString(schstring); SetPromptMsg(schstring);
             return(NotMatched);
           }
}
/* ------------------------------------------------------------------------------ */
int Quote_StateDoc()
{
 QP=0; QCP=0; EState=State_Quote; PromptMessage(ShowQuotePrompt); return(OK);
}
/* ------------------------------------------------------------------------------ */
int Quote_StateQuote()
{
 int rep;
 PromptCur=strlen(QuotePrompt)+QP+1;
 if (PromptCur>WinWidth-1) { PromptCur=WinWidth-1; }
 EState=State_Doc;
 if      (QArrMatchedWith("abort")) { Abort_Quote(); }
 else if (QArrMatchedWith("q"))     { return(Quit_Quote()); }
 else if (QArrMatchedWith("s"))     { return(Save_Quote()); }
 else if (QArrPrefixedBy("w "))     { return(Write_Quote()); }
 else if (QArrPrefixedBy("a "))     { return(Append_Quote()); }
 else if (QArrPrefixedBy("i "))     { return(Input_Quote()); }
 else if (QArrPrefixedBy("!"))      { return(Escape_Quote()); }
 else                               { return(MoveLineNumberOrError_Quote()); }
}
/* ------------------------------------------------------------------------------ */
Abort_Quote()
{
 PromptMessage(EditAborted); SysExit(OK);
}
/* ------------------------------------------------------------------------------ */
Quit_Quote()
{
 int rep;   OP=Quit;
 if   (DocChanged) { if (Backup==TRUE) { WriteBackupFile(); }
                     rep=(WriteToStore(FName,"w",1,Max+LP-RP));
                     if   (rep==OK) { SysExit(rep); }
                     else           { return(rep); }
                   }
 else              { PromptMessage(DocNotChanged); SysExit(OK); }
}
/* ------------------------------------------------------------------------------ */
Save_Quote()
{
 int rep;  OP=Save;
 if   (DocChanged) { if (Backup==TRUE) { WriteBackupFile(); }
                     rep=(WriteToStore(FName,"w",1,Max+LP-RP)); DocChanged=FALSE;
                     return(rep);
                   }
 else              { return(DocNotChanged); }
```

```
}
/* --------------------------------------------------------------------------- */
Write_Quote()
{
 char filename[QMax];   OP=Write;
 if    (MP!=(-1) && MP!=CP) { CopyQBuffToString(filename,2); SysTranslate(filename);
                              PromptCur=1;
                              if    (MP<CP) { return(WriteToStore(filename,"w",MP+1,CP)); }
                              else          { return(WriteToStore(filename,"w",CP+1,MP)); }
                            }
 else                       { return(NoTextMarked); }
}
/* --------------------------------------------------------------------------- */
int Append_Quote()
{
 char filename[QMax];   OP=Append;
 if    (MP!=(-1) && MP!=CP) { CopyQBuffToString(filename,2); SysTranslate(filename);
                              PromptCur=1;
                              if    (MP<CP) { return(WriteToStore(filename,"a",MP+1,CP)); }
                              else          { return(WriteToStore(filename,"a",CP+1,MP)); }
                            }
 else                       { return(NoTextMarked); }
}
/* --------------------------------------------------------------------------- */
int Input_Quote()
{
 struct stat stbuf;    char filename[QMax];   int ctrlfound=FALSE;   int rep;
 OP=Input; OPType=LeftInsert; CopyQBuffToString(filename,2); SysTranslate(filename);
 if (FileExists(filename,&stbuf))
       { if (NotDirectory(&stbuf))
               { if (ReadPermission(filename))
                       { rep=(ReadFromStore(filename,&stbuf,&ctrlfound));
                         if (ctrlfound) { PromptMessage(CntrlFound); }
                         if (rep==OK) { DocChanged=TRUE; }
                         return(rep);
                       }
                 else
                       { SetPromptMsg(filename); return(NoReadPermission); }
               }
         else
               { SetPromptMsg(filename); return(Directory); }
       }
 else
       { SetPromptMsg(filename); return(FileNotExist); }
}
/* --------------------------------------------------------------------------- */
int Escape_Quote()
{
 char command[QMax];   int c='\0';
 OP=Escape; CopyQBuffToString(command,1);
 if (strlen(command)!=0)
       { PosImage(); SetSysCursor(); CursorToNextLine(); ResetTerminal();
         system(command); GetTermCapAndSetWin();  ReadTermMode(); SetTerminal();
         PromptMessage(HitKeyToResume); while (c!=nl) { c=GetNextChar(); }
         RefreshDisplay_Doc9(); return(OK);
       }
 else
       { return(NoCommandGiven); }
}
/* --------------------------------------------------------------------------- */
int MoveLineNumberOrError_Quote()
{
 int numstring[QMax];   int lineX;   int prevCP=CP;
 CopyQBuffToString(numstring,0);
```

```
 if (CnvStringToNum(numstring,&lineX))
       { if (lineX>DocNL+1) { lineX=DocNL+1; }
         if   (lineX<CurY) { lineX=CurY-lineX; OP=LeftMoveLine; }
         else              { lineX=lineX-CurY; OP=RightMoveLine; }
         while (lineX!=0) { OP_Doc1(); lineX--; }
         OP=LeftMoveLine; DP_Doc1();
         if (CP>prevCP) { OPType=RightMove; } else { OPType=LeftMove; }
         OP=MoveLineNumberOrError; PromptDisplayed=FALSE; PosImage(); return(OK),
       }
 else
       { OP=MoveLineNumberOrError; return(QuoteError); }
}
/* ------------------------------------------------------------------------------------ */
int InsertChar_Quote()
{
 int ptr=QP;
 if   (QP!=QMax)
          { if   (OPChar!=TAB) { while (ptr!=QCP) { QArr[ptr+1]=QArr[ptr]; ptr--; }
                                                  QArr[QCP+1]=OPChar; QCP++; QP++;
                                                  return(OK);
                                                 }
            else                { return(IllegalQuoteChar); }
          }
 else
          { return(FullQuote); }
}
/* ------------------------------------------------------------------------------------ */
int LeftMoveChar_Quote()
{
 if   (QCP!=0) { QCP--; return(OK); }
 else          { return(TopQuote); }
}
/* ------------------------------------------------------------------------------------ */
int RightMoveChar_Quote()
{
 if   (QCP!=QP) { QCP++; return(OK); }
 else           { return(BotQnote); }
}
/* ------------------------------------------------------------------------------------ */
int LeftDeleteChar_Quote()
{
 int ptr=QCP;
 if   (QCP!=0) { while (ptr!=QP) { QArr[ptr]=QArr[ptr+1]; ptr++; }
                 QCP--; QP--; return(OK);
               }
 else          { return(TopQuote); }
}
/* ------------------------------------------------------------------------------------ */
int RightDeleteChar_Quote()
{
 int ptr=QCP+1;
 if   (QCP!=QP) { while (ptr!=QP) { QArr[ptr]=QArr[ptr+1]; ptr++; }
                  QP--; return(OK);
                }
 else           { return(BotQuote); }
}
/* ------------------------------------------------------------------------------------ */
int QArrPrefixedBy(target)   char target[];
{
 int prefixed;    int ptr=0;    int length=strlen(target);
 prefixed=(QP>=length);
 while (prefixed && ptr!=length) { prefixed=(Lower(QArr[ptr+1])==target[ptr]); ptr++; }
 return(prefixed);
}
```

```
/* ---------------------------------------------------------------------------------- */
int QArrMatchedWith(target)   char target[];
{
 int matched;    int ptr=0;
 matched=(QP==strlen(target));
 while (matched && ptr!=QP) { matched=(Lower(QArr[ptr+1])==target[ptr]); ptr++; }
 return(matched);
}
/* ---------------------------------------------------------------------------------- */
CopyQBuffSBuff()
{
 SP=0; while (SP!=QP) { SP++; SArr[SP]=QArr[SP]; }
}
/* ---------------------------------------------------------------------------------- */
CopyQBuffRpBuff()
{
 RpP=0; while (RpP!=QP) { RpP++; RpArr[RpP]=QArr[RpP]; }
}
/* ---------------------------------------------------------------------------------- */
CopyQBuffToString(string,ptr)   char *string;   int ptr;
{
 while (ptr!=QP) { *string=QArr[ptr+1]; string++; ptr++; }    *string='\0';
}
/* ---------------------------------------------------------------------------------- */
CopySBuffToString(string)   char *string;
{
 int ptr=0; while (ptr!=SP) { *string=SArr[ptr+1]; if (*string==nl) { *string='|'; }
                               string++; ptr++;
                            }
 *string='\0';
}
/* ---------------------------------------------------------------------------------- */
CopyRBuffToString(string)   char *string;
{
 int ptr=0; while (ptr!=RpP) { *string=RpArr[ptr+1]; if (*string==nl) { *string='|'; }
                                string++; ptr++;
                             }
 *string='\0';
}
/* ---------------------------------------------------------------------------------- */
int GetSBuffNL()
{
 int NumNL=0;   int SPptr=0;
 while (SPptr!=SP) { SPptr++; if (SArr[SPptr]==nl) NumNL++; }
 return(NumNL); }

/* ---------------------------------------------------------------------------------- */
int GetRpBuffNL()
{
 int NumNL=0;   int RpPptr=0;
 while (RpPptr!=RpP) { RpPptr++; if (RpArr[RpPptr]==nl) NumNL++; }
 return(NumNL); }

/* ---------------------------------------------------------------------------------- */



                         /* IMPLEMENTATION OF DDC9 */
                       /* ======================== */
```

```
#include <stdio.h>
#include "/se/daven/ox/c/c2/externglobals.c"

/* -------------------------------------------------------------------------------- */
int OP_Doc9()
{
 int prevDocNL=DocNL;  int prevCP=CP;   int prevLP=LP;   int prevRP=RP;   int prevQP=QP;
 int prevCurY=CurY;    int first;       int last;        int temp;        int rep;
 if       (OP==NotImplemented)
            { return(OPNotImplemented); }
 else if (OP==CentreWindow)
            { return(CentreWindow_Doc9()); }
 else if (OP==RefreshDisplay)
            { return(RefreshDisplay_Doc9()); }
 else if (OP==CursorUpPage)
            { return(Screen_UpPage()); }
 else if (OP==CursorDownPage)
            { return(Screen_DownPage()); }
 else if (OP==Replace)
            { return(Screen_Replace(prevCurY)); }
 else if (OP==ShowDocStats)
            { return(Screen_ShowDocStats()); }
 else
            { if  ((rep=OP_Doc8())!=OK)
                    { return(rep); }
              else
                    {  if (EState==State_Quote)
                              { if (prevQP!=QP) DisplayQuoteBuffer(); }
                       else
                              { CheckFlashBrackets();
                                first=prevCurY-WinOffY;
                                last=first+DocNL-prevDocNL+NLRem-NLIns;
                                if (first>last)      { temp=first; first=last; last=temp; }
                                WindowPolicy(first,last);
                              }
                       return(OK);
                     }
            }
}
/* -------------------------------------------------------------------------------- */
int Initialize_Doc9()
{
 int ptr=0;   Initialize_Doc8();
 CLSAdjust(); PromptCur=WinWidth; PosImage(); WinOffX=0; WinOffY=0; SetDocCursor();
 while (ptr!=WinHeight) { WinLineOK[ptr+1]=FALSE; ptr++; }
}
/* -------------------------------------------------------------------------------- */
int CentreWindow_Doc9()
{
 if (EState==State_Doc)
       { if       (CurY-WinOffY>=HalfWinHeight)
                    { OPType=RightMove; MoveWindowDown(CurY-WinOffY-HalfWinHeight);
                      SetDocCursor(); return(OK);
                    }
         else if (CurY-WinOffY<HalfWinHeight && CurY>=HalfWinHeight)
                    { OPType=LeftMove; MoveWindowUp(HalfWinHeight-CurY+WinOffY);
                      SetDocCursor(); return(OK);
                    }
         else
                    { return(TooNearTop); }
       }
 else
      { return(OP_Doc8()); }
```

```
}
/* ---------------------------------------------------------------------------- */
int RefreshDisplay_Doc9()
{
 CLSAdjust(); return(OK);
}
/* ---------------------------------------------------------------------------- */
Screen_UpPage()
{
 int rep=OP_Doc8().
 if (rep==OK)
       {  if      (WinOffY>=PageHeight)             { MoveWindowUp(PageHeight); }
          else if (WinOffY<PageHeight && WinOffY!=0) { MoveWindowUp(WinOffY); }
       }
 return(rep);
}
/* ---------------------------------------------------------------------------- */
Screen_DownPage()
{
 int rep=OP_Doc8();
 if (rep==OK) { MoveWindowDown(PageHeight); }
 return(rep);
}
/* ---------------------------------------------------------------------------- */
Screen_Replace(prevCurY)   int prevCurY;
{
 int SBuffNL;  int RpBuffNL;   int rep;
 if ((rep=OP_Doc8())==OK)
      { SBuffNL=GetSBuffNL(); RpBuffNL=GetRpBuffNL();
        if   (SBuffNL==RpBuffNL)
               { WindowLinesBad(prevCurY-WinOffY,prevCurY-WinOffY+SBuffNL); }
        else
               { WindowLinesBad(prevCurY-WinOffY,WinHeight); }
        if (SP<RpP) { OPType=RightMove; } else { OPType=LeftMove; }
        WindowPolicy();
      }
 return(rep);
}
/* ---------------------------------------------------------------------------- */
int Screen_ShowDocStats()
{  int ptr=0;
 if (EState==State_Doc)    { PromptMessage(ShowStats); PromptDisplayed=FALSE; return(OK); }
 else                      { return(OP_Doc8()); }
}
/* ---------------------------------------------------------------------------- */
CheckFlashBrackets()
{
 int prevCP=CP;   int prevStartln=Startln;   char closebracket=OPChar;  int count=0;
 int NumNL=0;    int nomatch=TRUE;          char openbracket;          char arrchar;
 int x;          int y;
 if (OP==InsertChar &&
     (OPChar=='}' || OPChar==']' || OPChar==')' || OPChar=='>') &&
      Not(CharAvailable()))
       { if      (closebracket=='}')  { openbracket='{'; }
         else if (closebracket==']')  { openbracket='['; }
         else if (closebracket==')')  { openbracket='('; }
         else                         { openbracket='<'; }
         while (CP>WinStartln+WinOffX+1 && nomatch)
               { CP--; arrchar=GetArrCont(CP);
                 if      (arrchar==nl)
                           { NumNL++; }
                 elee if (arrchar==closebracket)
                           { count++; }
                 else if (arrchar==openbracket)
```

```
                          { if (count>0)
                                { count--; }
                            else
                                { SetStartln(); x=CP-Startln+1-WinOffX;
                                  y=CurY-NumNL-WinOffY; nomatch=FALSE;
                                  if (x-1>=1 && x-1<=WinWidth &&
                                      y>=1 && y<=WinHeight)
                                        { WindowLinesBad(y+NumNL,y+NumNL);
                                          DisplayWindowRange(y+NumNL,y+NumNL);
                                          fflush(stdout);
                                          SetTermCursor(x-1,y); fflush(stdout);
                                          delay(); CP=prevCP; Startln=prevStartln;
                                        }
                                }
                          }
                  }
        CP=prevCP; Startln=prevStartln;
      }
}
/* ------------------------------------------------------------------------- */
SetWinStartln()
{
 register int numnl=CurY-WinOffY-1;
 WinStartln=Startln;  while (numnl!=0) { WinStartln=GetPrevStartln(WinStartln); numnl--; }
}
/* ------------------------------------------------------------------------- */
int GetPrevStartln(ptr)   register int ptr;
{
 ptr--;  while (ptr!=0 && GetArrCont(ptr)!=nl) { ptr--; }  return(ptr);
}
/* ------------------------------------------------------------------------- */
int GetNextStartln(ptr)   register int ptr;
{
 if (ptr!=Max+LP-RP) { ptr++;
                       while (ptr!=Max+LP-RP && GetArrCont(ptr)!=nl) { ptr++; }
                     }
 return(ptr);
}
/* ------------------------------------------------------------------------- */
int DisplayWindowLine(startlnptr)   register int startlnptr;
{
 register int x=0;
 while (startlnptr+x!=Max+LP-RP && GetArrCont(startlnptr+x+1)!=nl && x!=WinOffX)
        { x++; }
 while (startlnptr+x!=Max+LP-RP && GetArrCont(startlnptr+x+1)!=nl && x!=WinOffX+WinWidth)
        { putchar(GetArrCont(startlnptr+x+1)); x++; }
 if (x!=WinOffX+WinWidth) { ClearToEndOfLine(); }
}
/* ------------------------------------------------------------------------- */
DisplayWindowRange(first,last)   register int first;   register int last;
{
 register int ptr=WinStartln;   register int y=first;
  while (y!=1) { ptr=GetNextStartln(ptr); y--; }
  while (first<=last && Not(CharAvailable()) && NoInterrupt)
         { if (Not(WinLineOK[first]))
                 { SetTermCursor(1,first); DisplayWindowLine(ptr);
                   WinLineOK[first]=TRUE;
                 }
           ptr=GetNextStartln(ptr); first++;
         }
}
/* ------------------------------------------------------------------------- */
WindowLinesBad(first,last)   int first;   int last;
{
```

```
 if      (first<1)       { first=1; }
 else if (last>WinHeight) { last=WinHeight; }
 while (first<=last) { WinLineOK[first]=FALSE; first++; }
}
/* ---------------------------------------------------------------------------------- */
DisplayTheWindow()
{
 DisplayWindowRange(1,WinHeight);
}
/* ---------------------------------------------------------------------------------- */
DisplayCurLine()
{
 SetTermCursor(1,CurY-WinOffY); DisplayWindowLine(Startln);
}
/* ---------------------------------------------------------------------------------- */
MoveWindowDown(y,first)   int y;   int first;
{
 WinOffY=WinOffY+y;  SetWinStartln();
 if (OPType==RightMove)
       { if    (y<=HalfWinHeight+1) { ScrollUpAdjust(y); }
         else                       { WindowLinesBad(1,WinHeight); }
       }
 else
       { if (y+WinHeight-first<=HalfWinHeight+1) { WindowLinesBad(first,WinHeight);
                                                    ScrollUpAdjust(y);
                                                  }
         else                                     { WindowLinesBad(1,WinHeight); }
       }
}
/* ---------------------------------------------------------------------------------- */
MoveWindowUp(y,last)   int y;   int last;
{
 WinOffY=WinOffY-y;  SetWinStartln();
 if (OPType==LeftMove) { if   (y<=HalfWinHeight+1) { ScrollDownAdjust(y,1); }
                          else                      { WindowLinesBad(1,WinHeight); }
                       }
 else                  { if (last<=HalfWinHeight+1)
                              { WindowLinesBad(1,last);
                                if (last<=CurY-WinOffY)
                                     { ScrollDownAdjust(CurY-WinOffY-last,last+1); }
                                else
                                     { ScrollUpAdjust(last-CurY+WinOffY); }
                              }
                         else
                                 { WindowLinesBad(1,WinHeight); }
                       }
}
/* ---------------------------------------------------------------------------------- */
MoveWindowLeft(x)   int x;
{
 WinOffX=WinOffX-x; WindowLinesBad(1,WinHeight);
}
/* ---------------------------------------------------------------------------------- */
MoveWindowRight(x)   int x;
{
 WinOffX=WinOffX+x; WindowLinesBad(1,WinHeight);
}
/* ---------------------------------------------------------------------------------- */
WindowPolicy(first,last)   int first;   int last;
{
 int winx=CurX-WinOffX;   int winy=CurY-WinOffY;
 if   (winx>=1 && winx<=WinWidth)
                  { if   (winy>=1 && winy<=WinHeight) { CursorInWindow(first,last); }
                    else                              { Scroll(first,last); }
```

```
                }
 else
                { if   (winy>=1 && winy<=WinHeight) { Pan(); }
                  else                              { ScrollAndPan(); }
                }
}
/* ---------------------------------------------------------------------------- */
Scroll(first,last)   int first;   int last;
{
 int winy=CurY-WinOffY;
 if       (winy<1 && CurY>=QtrWinHeight)
            { MoveWindowUp(WinOffY-CurY+QtrWinHeight,last); }
 else if (winy<1 && CurY<QtrWinHeight)
            { MoveWindowUp(WinOffY,last); }
 else
            { MoveWindowDown(CurY-PageHeight-WinOffY,first); }
}
/* ---------------------------------------------------------------------------- */
Pan()
{
 int winx=CurX-WinOffX;
 if       (winx<1 && CurX>=QtrWinWidth) { MoveWindowLeft(WinOffX-CurX+QtrWinWidth); }
 else if (winx<1 && CurX<QtrWinWidth)  { MoveWindowLeft(WinOffX); }
 else                                   { MoveWindowRight(CurX-PageWidth-WinOffX); }
}
/* ---------------------------------------------------------------------------- */
ScrollAndPan()
{
 int winy=CurY-WinOffY;
 if       (winy<1 && CurY>=QtrWinHeight) { WinOffY=CurY-QtrWinHeight; }
 else if (winy<1 && CurY<QtrWinHeight)  { WinOffY=0; }
 else                                    { WinOffY=CurY-PageHeight; }
 SetWinStartln(); Pan();
}
/* ---------------------------------------------------------------------------- */
CursorInWindow(first,last)   int first;   int last;
{
 if (OPType!=LeftMove && OPType!=RightMove && OPType!=NoMove)
       { if (first==last)
              { DisplayCurLine(); }
         else
              { if (last>HalfWinHeight || last>=DocNL+1-WinOffY)
                      { WindowLinesBad(first,WinHeight);
                        DisplayWindowRange(first,WinHeight); }
                else
                      { if (OPType==LeftInsert || OPType==RightInsert)
                                   { WindowLinesBad(first,last);
                                     ScrollDownAdjust(last-first,last+1);
                                   }
                        else if (OPType==LeftDelete || OPType==RightDelete)
                                   { WindowLinesBad(1,last+last-first);
                                     ScrollUpAdjust(last-first);
                                   }
                      }
              }
       }
}
/* ---------------------------------------------------------------------------- */
DisplayQuoteBuffer()
{
 register int ptr=0;   char c;
 SetInitQuoteCursor();
 while (ptr!=QP && ptr!=WinWidth-QuotePromptLength-1)
     { c=QArr[ptr+1];
```

```
        if   (c==nl) { putchar('|'); }
        else         { putchar(QArr[ptr+1]); }
        ptr++;
      }
  while (ptr!=WinWidth-QuotePromptLength-1) { putchar(sp); ptr++; } SetQuoteCursor();
}
/* -------------------------------------------------------------------------------- */
```

Appendix C

Implementation Of The Editor Specification (Continued)

```
                                  /* main.c */
                                /* ========== */

#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <sys/fcntl.h>
#include "/se/daven/ox/c/c2/globals.c"
#include "/se/daven/ox/c/c1/read.c"
#include "/se/daven/ox/c/c1/write.c"
#include "/se/daven/ox/c/c2/screen.c"
#include "/se/daven/ox/c/c2/term.c"
#include "/se/daven/ox/c/c2/sysentryexit.c"
#include "/se/daven/ox/c/c2/messages.c"
#include "/se/daven/ox/c/c2/utilities.c"
/* ------------------------------------------------------------------------------------------ */
main(args,argval)    int args;    char *argval[];

     /* "args" is the number of arguments present in system command line    */
     /* "argval" is the array of argument addresses in system command line */
{
 int InitialLine=1;   int cntrlfound=FALSE;   int newfile=FALSE;   int rep;
 Backup=FALSE; TerminalSet=FALSE; ScreenCleared=FALSE; Kbdptr=0; NoInterrupt=TRUE;
 GetTermCapAndSetWin(); ReadTermMode(); SetTerminal(); SysEntry(args,argval,&InitialLine);

GetNextChar();

 rep=(StartEditFile(FName,&cntrlfound,&newfile));
 if   (rep!=OK) { SysExit(rep,FName); }
 else           { OP=MoveToTop; OP_Doc8();
                  while (InitialLine!=1) { OP=RightMoveLine; OP_Doc8(); InitialLine--;}
                  if (CurY>WinHeight) { WinOffY=CurY-PageHeight; SetWinStartLn(); }
                  DisplayTheWindow();
                  if   (newfile)  { PromptMessage(EditingNewFile); PromptDisplayed=FALSE;}
                  else            { PromptMessage(EditingFile); PromptDisplayed=TRUE;}
                  if (cntrlfound) { PromptMessage(CntrlFound); }
                  SetDocCursor(); fflush(stdout);
                }
 while (TRUE) { DPType=NoMove; ExecuteCommand();
                if (EState==State_Doc)
                      { if (Not(CharAvailable())) { DisplayTheWindow(); }
                        if (Not(NoInterrupt)) { PromptMessage(Interrupted);
                                                ClearInterrupt();
                                              }
                        SetDocCursor();
                      }
                else
                      { SetQuoteCursor(); }
                fflush(stdout);
              }
}
/* ------------------------------------------------------------------------------------------ */
```

```
ExecuteCommand()
{
 int rep=OP_Doc9(KbdRead());
 if (EState==State_Doc)
       { if (rep==OK)
              { if (Not((PromptDisplayed)) && OP!=Quote && OP!=UpSearch &&
                    OP!=DownSearch && OP!=Replace && OP!=Input && OP!=Save &&
                    OP!=Write && OP!=ShowDocStats)
                        { PromptMessage(EditingFile); }
              }
         else
              { PromptMessage(rep); }
         /* SetDocCursor(); */
       }
 else
       { if (rep!=OK) { RingBell(); } }
 /* fflush(stdout); */
}
/* ------------------------------------------------------------------------------ */
```

```
                              /* read.c */
                             /* ========== */

/* ------------------------------------------------------------------------------ */
int StartEditFile(filename,ctrlfound,newfile)   char *filename;   int *ctrlfound;
                                                int *newfile;
   /* "filename" is a pointer to a file. Checks to see if edit possible (if the file  */
   /* either doesn't exist and can be created, or if it exists then it must read      */
   /* permission and must not exceed the editor's capacity). Returns appropriate rep. */
   /* and sets *cntrlfound to TRUE if a control character read (which is discarded)   */
{
 int rep;  struct stat stbuf;
 if (FileExists(filename,&stbuf))
       { if (NotDirectory(&stbuf))
              { if (ReadPermission(filename))
                     { Initialize_Doc9(); OP=Input; DPType=LeftInsert;
                       rep=ReadFromStore(filename,&stbuf,ctrlfound);
                       Update_Doc3(0,0,Max); Update_Doc4(); Update_Doc6(0);
                       return(rep);
              }
                else
                     { SysExit(NoReadPermission); }
              }
         else
              { SysExit(Directory); }
       }
 else
       { if (WritePermission(filename))
              { Initialize_Doc9(); ScreenCleared=TRUE; *newfile=TRUE; return(OK); }
         else
              { SysExit(NoWritePermission); }
       }
}
/* ------------------------------------------------------------------------------ */
int FileExists(filename,stbufptr)   char *filename;   struct stat *stbufptr;

  /* "filename" points to the file. "stbufptr" is a pointer to stbuf. A "stat" call */
```

```
  /* is attempted, and, if successful, TRUE returned, else FALSE returned.         */
{
 return((stat(filename,stbufptr)!=-1));
}
/* ------------------------------------------------------------------------------ */
int NotDirectory(stbufptr)   struct stat *stbufptr;

  /* "filename" points to the file. "stbufptr" is a pointer to stbuf. If file not */
  /* a directory, function returns TRUE, else returns FALSE                       */
{
 return((stbufptr->st_mode & S_IFDIR) ^ S_IFDIR);
}
/* ------------------------------------------------------------------------------ */
ReadPermission(filename)   char *filename;

      /* "filename" a pointer to a file. If file has read permission, function */
      /* returns TRUE, else returns FALSE                                      */
{
 FILE *fopen(),*SysPtr;
 SysPtr=fopen(filename,"r");
 if (SysPtr!=NullPtr) { fclose(SysPtr); return(TRUE); }
 else                 { return(FALSE); }
}
/* ------------------------------------------------------------------------------ */
WritePermission(filename)   char *filename;

           /* "filename" is a pointer to a filename. If file can be written to */
           /* function returns TRUE, else returns FALSE                        */
{
 FILE *fopen(),*SysPtr;
 if ((SysPtr=fopen(filename,"w"))!=NullPtr     )          /* if file can be opened ... */
        { fclose(SysPtr); unlink(filename); return(TRUE); } /* ... close and delete it   */
 else                                                     /* else file can't be opened */
        { return(FALSE); }
}
/* ------------------------------------------------------------------------------ */
int ReadFromStore(filename,stbufptr,ctrlfound)  char *filename:    struct stat *stbufptr;
                                                int *ctrlfound;

      /* If control characters found, discarded and *ctrlfound set to TRUE  */
      /* except for TAB controls, which are expanded in "ExpandTabs".       */
{
 int prevLP=LP;   int lineX=0;   int NoReadError=TRUE;   int cntrlfnd;   int x;
 FILE *fopen(),*SysPtr;        int filelength=FileLength(stbufptr);
 SetPromptMsg(filename);
 if   (filelength<=RP-LP)
        { if ((SysPtr=fopen(filename,"r"))!=NullPtr)
                 { ScreenCleared=TRUE; Standardize_Doc1(); PromptMessage(ReadingFile);
                   while (LP<RP && NoReadError && filelength>0 && NoInterrupt)
                         { x=getc(SysPtr); filelength--;
                           cntrlfnd=ControlChar(x);
                           while (cntrlfnd && filelength>0  && NoInterrupt)
                                 { *ctrlfound=TRUE; x=getc(SysPtr); filelength--;
                                   cntrlfnd=ControlChar(x);
                                 }
                           if (x==LF || x==RET) { x=nl; }
                           Arr[LP+1]=x; LP++; lineX++;
                           if      (x==nl)    { lineX=0; StripTrailWS(); }
                           else if (x==TAB)  { ExpandTabs(&lineX); }
                           NoReadError=(x!=EOF && NoInterrupt);
                         }
                   fclose(SysPtr);
                   if      (LP==RP)           { LP=prevLP; return(Full_Tabs); }
                   else if (Not(NoReadError)) { LP=prevLP; return(ReadError); }
```

```
            else                       { CP=LP; PromptMessage(Done); return(OK); }
          }
        else
          { return(CannotOpenFile); }
      }
 else
      { return(Full); }
}
/* ---------------------------------------------------------------------------------- */
int FileLength(stbufptr)   struct stat *stbufptr;

          /* "stbufptr" is a pointer to "stbuf" in StartEditFile */
{
 return(stbufptr->st_size);
}
/* ---------------------------------------------------------------------------------- */
int ControlChar(x)   int x;
{
 return((x<' ' || x>'~') && x!=RET && x!=TAB && x!=LF);
}
/* ---------------------------------------------------------------------------------- */
ExpandTabs(ptrlineX)   int *ptrlineX;
{
 Arr[LP]=sp; (*ptrlineX)=tabstop-((*ptrlineX)%tabstop);
 if (*ptrlineX==8) { *ptrlineX=0; }
 while ((*ptrlineX)!=0 && LP<RP) { Arr[LP+1]=sp; LP++; (*ptrlineX)--; }
}
/* ---------------------------------------------------------------------------------- */
StripTrailWS()
{
 while(LP>CP+1 && Arr[LP-1]==sp) { LP--; Arr[LP]=nl; }
}
/* ---------------------------------------------------------------------------------- */
```

```
                                /* write.c */
                               /* ========== */

/* ---------------------------------------------------------------------------------- */
int WriteToStore(filename,filemode,first,last)   char filename[];   char filemode[];
                                                  int  first;        int  last;
{
 FILE *fopen(),*SysPtr;    int NoWriteError=TRUE;    int ptr=first-1;
 SysTranslate(filename); SysPtr=fopen(filename,filemode);
 SetPromptMsg(filename);
 if (SysPtr!=NullPtr)
     { PromptMessage(WritingFile);
       while (NoWriteError && ptr!=last)
              { NoWriteError=((putc(GetArrCont(ptr+1),SysPtr)!=EOF) && NoInterrupt);
                ptr++;
              }
       if (NoWriteError && GetArrCont(ptr)!=nl)
            { NoWriteError=((putc(nl,SysPtr)!=EOF) && NoInterrupt); }
       fclose(SysPtr);
       if    (NoWriteError) { PromptMessage(Done); return(OK); }
       else                 { return(WriteError); }
     }
 else
```

```
      { return(CannotOpenFile); }
}
/* -------------------------------------------------------------------------------- */
void WriteBackupFile()

      /* FName++ (FName with "++" appended) is first unlinked, and, if possible, */
      /* FName++ is linked to FName. An appropriate PromptMessage is sent.       */
{
 char backupFName[FNameMaxx];
 strcpy(backupFName,FName); strcat(backupFName,"++");
 SetPromptMsg(backupFName); PromptMessage(UpdatingBackup);
 unlink(backupFName);
 if (link(FName,backupFName)!=-1)                         /* new link can be made */
      { unlink(FName); }
 else                                                   /* new link can't be made */
      { PromptMessage(CannotUpdateBackup); }
}
/* -------------------------------------------------------------------------------- */
```

```
                              /* globals.c */
                            /* ============== */

#include "consts.c"
/* -------------------------------------------------------------------------------- */
int     LP,RP,CP;                                                    /* ConcDoc1 */
char    Arr[Maxx];

int     CurX,CurY,StartIn,EndIn,DocNL,WSRem,NLRem;                   /* ConcDoc3 */

int     WSIns,NLIns,PageWidth,PageHeight;                            /* ConcDoc4 */

int     MP,PP;                                                       /* ConcDoc6 */
char    PArr[Maxx];

char    QArr[QMaxx],SArr[QMaxx],RpArr[QMaxx];                        /* ConcDoc8 */
int     QP,QCP,SP,RpP;
int     MatchedLength;
char    FName[FNameMaxx];
int     FP;
int     EState;

int     WinOffX,WinOffY;                                             /* ConcDoc9 */
int     WinStartIn;
int     WinLineOK[MaxWinHeight+2];
int     WinHeight,WinWidth;
int     HalfWinHeight,HalfWinWidth;
int     QtrWinHeight,QtrWinWidth;

char    KbdArr[kbdMaxx];                                            /* kbdread.c */
int     Kbdptr;
int     HomePath;
int     Key;
int     OP;
int     OPType;
char    OPChar;
int     OPArr[kbdMaxx];
```

```
int     NoInterrupt;                                              /* sysentryexit.c */

int     Backup;                                                   /* write.c */

int     PromptDisplayed;                                          /* for prompt display */
int     DocChanged;
int     PromptCur;
char    PromptMsg[FNameMaxx];

short   ospeed;                                                   /* term.c */
int     TermType;
struct  sgttyb Sgttyb;
int     fcntlFlag;
int     TerminalSet;
int     ScreenCleared;

char    PC;                                       /* termcap padding character    */
char    CM[cntrlsize];                            /* termcap cursor motion        */
char    RC[cntrlsize];                            /* termcap carriage return      */
char    OD[cntrlsize];                            /* termcap down one line        */
char    SO[cntrlsize];                            /* termcap begin stand-out mode */
char    SE[cntrlsize];                            /* termcap end stand-out mode   */
char    CE[cntrlsize];                            /* termcap clear to end of line */
char    BL[cntrlsize];                            /* termcap ring bell            */
char    SR[cntrlsize];                            /* termcap reverse scroll       */
char    CL[cntrlsize];                            /* termcap clear entire screen  */
char    BC[cntrlsize];                            /* termcap backspace            */
char    UP[cntrlsize];                            /* termcap cursor up            */

/* -------------------------------------------------------------------------------- */
```

```
                                  /* term.c */
                               /* ============ */
/* -------------------------------------------------------------------------------- */
GetTermCapAndSetWin()

   /* Loads "tcEntry" buffer from termcap. SysExit if terminal not defined or        */
   /* termcap not found, else termcap values initialized Sets WinWidth and WinHeight */
   /* using "ioctl" call, else termcap, else DefaultWinWidth and DefaultWinHeight;   */
   /* if larger than MaxWinWidth or MaxWinHeight, then SysExit.                      */
{
 struct ttysize ttysz;   char *ptr[cntrlsize];    char tcEntry[tcsize];
 char temp[cntrlsize];
 if ((TermType=getenv("TERM"))==0) { SysExit(NoTERM); }
 switch(tgetent(tcEntry,TermType)) { case  0 :  SysExit(UnknownTerminal);
                                     case -1 :  SysExit(NoTermcapFile);
                                   }
 *ptr=CM; if (tgetstr("cm",ptr)==0) SysExit(InadeqTermCap,"cm");
 *ptr=RC; if (tgetstr("cr",ptr)==0) { RC[0]=CR; RC[1]=0; }
 *ptr=OD; if (tgetstr("do",ptr)==0) { OD[0]=LF; OD[1]=0; }
 *ptr=SO; if (tgetstr("so",ptr)==0) if (tgetstr("md",ptr)==0) SysExit(InadeqTermCap,"so");
 *ptr=SE; if (tgetstr("se",ptr)==0) if (tgetstr("mg",ptr)==0) SysExit(InadeqTermCap,"se");
 *ptr=CE; if (tgetstr("ce",ptr)==0) SysExit(InadeqTermCap,"ce");
 *ptr=BL; if (tgetstr("bl",ptr)==0) if (tgetstr("vb",ptr)==0) { BL[0]=BEL; BL[1]=0; }
 *ptr=SR; if (tgetstr("sr",ptr)==0) if (tgetstr("al",ptr)==0) SysExit(InadeqTermCap,"sr");
 *ptr=CL; if (tgetstr("cl",ptr)==0) SysExit(InadeqTermCap,"cl");
 *ptr=UP; if (tgetstr("up",ptr)==0) SysExit(InadeqTermCap,"up");
```

```
 BC[0]=BS;BC[1]=0;
 *ptr=BC; if (tgetflag("bs")==0) if (tgetstr("bc",ptr)==0) SysExit(InadeqTermCap,"bs");
 PC=PAD;
 *ptr=temp; if (tgetstr("pc",ptr)!=0) { PC=temp[0]; }
 if (ioctl(0,TIOCGSIZE,&ttysz)==-1 || ttysz.ts_lines<=0 || ttysz.ts_cols<=0)
      { ttysz.ts_cols=tgetnum("co");
        if (ttysz.ts_cols==-1)
              { SetSysCursor(); printf("Failed to get #columns of tty...setting default");
                ttysz.ts_cols=DefaultWinWidth;
              }
        ttysz.ts_lines=tgetnum("li");
        if (ttysz.ts_lines==-1)
              { SetSysCursor(); printf("Failed to get #lines of tty...setting default");
                ttysz.ts_lines=DefaultWinHeight;
              }
      }
 if (ttysz.ts_cols>MaxWinWidth)   { SysExit(TooManyColsTTY,ttysz.ts_cols); }
 if (ttysz.ts_lines>MaxWinHeight) { SysExit(TooManyRowsTTY,ttysz.ts_lines); }
 if (ttysz.ts_cols<=0) { SetSysCursor();
                         printf("Failed to get #columns in TTY ... setting default");
                       }
 WinWidth=ttysz.ts_cols;
 if (ttysz.ts_lines<=0) { SetSysCursor();
                          printf("Failed to get #lines in TTY ... setting default");
                        }
 WinHeight=ttysz.ts_lines;
 WinHeight--;                                                      /* for prompt line */
 if (WinHeight<MinWinHeight) { SysExit(WindowTooShort); }
 if (WinWidth<MinWinWidth)   { SysExit(WindowTooNarrow); }
 SetOtherWinHeights();
 return;
}
/* ------------------------------------------------------------------------------- */
SetOtherWinHeights()
{
 HalfWinHeight=(WinHeight/2); HalfWinWidth=(WinWidth/2);
 QtrWinHeight=(WinHeight/4); QtrWinWidth=(WinWidth/4);
 PageHeight=WinHeight-QtrWinHeight; PageWidth=WinWidth-QtrWinWidth;
}
/* ------------------------------------------------------------------------------- */
ReadTermMode()
{
 if   (ioctl(0,TIOCGETP,&Sgttyb)==0) { ospeed=Sgttyb.sg_ospeed; }
 else                                 { SysExit(SystemError,1,"ReadTermMode"); }
}
/* ------------------------------------------------------------------------------- */
void SetfcntlFlags()
{
 if ((fcntlFlag=fcntl(0,F_GETFL))==-1) SysExit(SystemError,1,"SetfcntlFlags");
 if (fcntl(0,F_SETFL,fcntlFlag | FNDELAY)==-1) SysExit(SystemError,1,"SetfcntlFlags");
}
/* ------------------------------------------------------------------------------- */
void ReSetfcntlFlags()
{
 if (fcntl(0,F_SETFL,fcntlFlag)==-1) SysExit(SystemError,1,"ReSetfcntlFlags");
}
/* ------------------------------------------------------------------------------- */
void SetTerminal()
{
 struct sgttyb SgttybTemp;   int SgttybMask=~(RAW+CRMOD+ECHO+LCASE+CBREAK+TANDEM);
 SgttybTemp=Sgttyb;
 SgttybTemp.sg_flags=((Sgttyb.sg_flags & SgttybMask) | (CBREAK+CRMOD) & (~XTABS));
 if (ioctl(0,TIOCSETP,&SgttybTemp)==-1) SysExit(SystemError,1,"SetTerminal");
 TerminalSet=TRUE;
```

```
 SetfcntlFlags();
 if (strcmp(TermType,"vt220")==0)
       { printf("%c%c",ESC,'=');                                    /* application keypad */
         printCSI(); printf("%c%d%c",'?',1,'h');                /* application cursor keys */
       }
}
/* ------------------------------------------------------------------------------------------- */
ResetTerminal()
{
 if (ioctl(0,TIOCSETP,&Sgttyb)==-1) SysExit(SystemError,1,"ResetTerminal");
 ReSetfcntlFlags();
}
/* ------------------------------------------------------------------------------------------- */
```

```
                                   /* sysentryexit.c */
                                 /* ================== */

/* ------------------------------------------------------------------------------------------- */
void SysEntry(args,argval,initline)   int  args;   char *argval[];   int *initline;

    /* "args" is the number of arguments present, "argval" an array of addresses. */
    /* "initline" a pointer to InitialLine (from main). function checks syntax    */
    /* of input - if bad, SysExit, but if OK, FName, InitialLine, Backup and      */
    /* numfilenames set using SetNextOption - if numfilenames not 1, SysExit.     */
{
 int numfilenames=0;
 argval++; args--;
 while (args>0) { if (*argval[0]=='-')                          /* option specified ... */
                        { SetNextOption(&args,&argval,initline,&numfilenames); }
                 else                                           /*... else the filename */
                        { if (strlen(*argval)>=FNameMax-2)
                               { SysExit(FilenameTooLong,*argval); }
                          strcpy(FName,*argval);
                          argval++; args--; numfilenames++;
                        }
                }
 if      (numfilenames==0) { SysExit(NoFilenameGiven); }
 else if (numfilenames>1)  { SysExit(TooManyFilenames); }
 LoadFredcap(); SetInterrupts();
}
/* ------------------------------------------------------------------------------------------- */
SetNextOption(as,av,i,f)   int  *as;   char **av[];
                           int  *i;    int     *f;

    /* "as" is a pointer to args, "av" points to the address of argval, "i" points to */
    /* InitialLine and "f" to numfilenames (all relating to SysEntry). The function   */
    /* takes the next argval, and, if recognized, (using the next argval if needed),  */
    /* sets the relevant flag. If the flag is not recognized, SysExit.                */

{
 if      (strcmp(**av,"-b")==0)                                 /* backup flag recognized */
    { Backup=TRUE; (*av)++; (*as)--; }
 else if (strcmp(**av,"-l")==0)                           /* initial line flag recognized */
    { if (*as>1)                                            /* another argument to take */
        { (*av)++; (*as)--;
          if   (CnvStringToNum(**av,i)) { (*av)++; (*as)--; }    /* a valid line number */
          else                          { SysExit(BadLineNumber,**av); }
```

```
        }
       else                                                    /* no line number present */
         { SysExit(BadCommandSyntax); }
     }
 else if (strcmp(**av,"-")==0)                              /* filename flag recognized */
     { if (*as>1)                                          /* another argument to take */
         { (*av)++; (*as)--;
           if (strlen(**av)>=FNameMax-2) { SysExit(FilenameTooLong,**av); }
           strcpy(FName,**av); (*av)++; (*as)--; (*f)++;
         }
      else
         { SysExit(NoFilenameGiven); }
     }
 else
     { SysExit(UnknownOption,**av); }
}
/* ---------------------------------------------------------------------------------- */
LoadFredcap()
{
 struct stat stbuf;   FILE *fopen(),*SysPtr;   int x;   int num;   int count;
 char filename[FNameMaxx];
 if ((HomePath=getenv("HOME"))==0) SysExit(NoHOMEset);
 strcpy(filename,"~/fredcap/$TERM"); SysTranslate(filename);
 if (Not(FileExists(filename,&stbuf))) { strcpy(filename,DefaultFredPath);
                                         SysTranslate(filename);
                                       }
 if (Not(FileExists(filename,&stbuf)))      { SysExit(NoFredcapFile); }
 if ((SysPtr=fopen(filename,"r"))==NullPtr) { strcpy(FName,filename);
                                              SysExit(CannotOpenFredcap);
                                            }
 Key=0; while (Key!=kbdMax) { OPArr[Key]=NotImplemented; Key++; }
 Key=0; x=getc(SysPtr);
 while (x!=EOF) { Key++;
                 while (x!=':' && x!=EOF) { x=getc(SysPtr); }
                 if (x!=EOF) { x=getc(SysPtr);
                              if (x!=sp && x!=nl && x!=EOF)
                                { num=0;
                                  while (IsDigitChar(x)) { num=(num*10)+(x-'0');
                                                           x=getc(SysPtr);
                                                         }
                                  OPArr[Key]=num;
                                }
                             }
                }
}
/* ---------------------------------------------------------------------------------- */
SysExit(rep,ptr,msg)   int rep;   int ptr;   char *msg;
{
 if (rep!=OK) { AbortMessage(rep,ptr); }
 SetSysCursor(); printf("\n");
 if (TerminalSet) { ResetTerminal(); }
 PosImage();
 if   (rep==SystemError) { perror(msg); printf("\n"); exit(stderr); }
 else                    { exit(rep); }
}
/* ---------------------------------------------------------------------------------- */
Interrupt()
{
 NoInterrupt=FALSE;
}
/* ---------------------------------------------------------------------------------- */
ClearInterrupt()
{
 NoInterrupt=TRUE;
```

```
}
/* ------------------------------------------------------------------------------------ */
Suspend()
{
 PosImage(); DisplayTheWindow(); SetSysCursor(); printf("\n");
 printf("%s Editing \"%s\"",EditorName,FName); fflush(stdout);
 if (TerminalSet) { ResetTerminal(); }
 sigsetmask(0); signal(SIGTSTP,SIG_DFL); kill(0,SIGTSTP);

 signal(SIGTSTP,Suspend); ClearInterrupt(); SetTerminal(); CLSAdjust(); DisplayTheWindow();
 if (EState==State_Doc)
       { PromptMessage(EditingFile); PromptMessage(Resumed); SetDocCursor(); }
 else
       { PromptMessage(ShowQuotePrompt); DisplayQuoteBuffer(); }
 fflush(stdout);
}
/* ------------------------------------------------------------------------------------ */
PanicExit()
{
 char filename[FNameMaxx];
 strcpy(filename,"~/fred.save"); SysTranslate(filename);
 if (DocChanged) { PanicWriteToStore(filename); }
 SetSysCursor(); printf("\n");
 if (TerminalSet) { ResetTerminal(); }
 PosImage();
 printf("Fatal interrupt received\n");
 if (DocChanged) { printf("Tried to save as \"%s\"\n",filename); }
 printf("\n"); exit(stderr);
}
/* ------------------------------------------------------------------------------------ */
int PanicWriteToStore(filename)   char *filename;
{
 int ptr=0;   FILE *fopen(),*SysPtr; SysPtr=fopen(filename,"w");
 if (SysPtr!=NullPtr)
      { while (ptr!=Max+LP-RP) { putc(GetArrCont(ptr+1),SysPtr); ptr++; }
        if (GetArrCont(ptr)!=nl) { putc(nl,SysPtr); }
        fclose(SysPtr);
      }
}
/* ------------------------------------------------------------------------------------ */
SetInterrupts()
{
 signal(SIGHUP,PanicExit);
 signal(SIGINT,Interrupt);
 signal(SIGQUIT,PanicExit);
 signal(SIGTSTP,Suspend);
}
/* ------------------------------------------------------------------------------------ */
SysTranslate(filename)   char *filename;
{
 int filelength=strlen(filename);   char newname[FNameMaxx];   int fptr=0;   int newptr=0;
 char var[FNameMaxx];   int val;
 while (fptr<filelength && newptr<FNameMax-2)
         { if      (filename[fptr]=='\\') { fptr++; newname[newptr]=filename[fptr];
                                            fptr++; newptr++;
                                          }
           else if (filename[fptr]=='~')  { fptr++;
                                            newptr=Insert(HomePath,newname,newptr);
                                          }
           else if (filename[fptr]=='$')  { fptr++;
                                            fptr=StripEnvVar(var,filename,fptr);
                                            if ((val=getenv(var))!=0)
                                                  { newptr=Insert(val,newname,newptr); }
                                          }
```

```
          else                          { newname[newptr]=filename[fptr];
                                          fptr++; newptr++;
                                        }
        }
 newname[newptr]='\0'; strcpy(filename,newname);
}
/* ------------------------------------------------------------------------ */
Insert(word,newname,newptr)   char *word;   char *newname;   int newptr;
{
 int wordlength=strlen(word);   int ptr=0;
 while(ptr<wordlength && newptr<FNameMax-2) { newname[newptr]=word[ptr]; ptr++; newptr++; }
 return(newptr);
}
/* ------------------------------------------------------------------------ */
StripEnvVar(var,filename,fptr)   char *var;   char *filename;   int fptr;
{
 int ptr=0;   char c=filename[fptr];
 while (IsAlphaNum(c)) { var[ptr]=c; fptr++; ptr++; c=filename[fptr]; }
 var[ptr]='\0';  return(fptr);
}
/* ------------------------------------------------------------------------ */
```

```
                              /* screen.c */
                            /* ============ */
/* ------------------------------------------------------------------------ */
DisplayChar(c)   int c;
{
/* if (c==sp) { c='_'; } else if (c==nl) { c='|'; } */
 putchar(c);
}
/* ------------------------------------------------------------------------ */
ScreenOutput(cntrlstr,lines)   register char *cntrlstr;   int lines;
{ tputs(cntrlstr,lines,DisplayChar); }
/* ------------------------------------------------------------------------ */
CLSAdjust()
{ ScreenOutput(CL,WinHeight+1); WindowLinesBad(1,WinHeight);
  PromptCur=WinWidth; PromptDisplayed=FALSE;     }
/* ------------------------------------------------------------------------ */
SetTermCursor(x,y)   int x;   int y;
{

 if (strcmp(tgoto(CM,x-1,y-1),"OOPS")==0) { SysExit(SystemError,1,"OOPS"); }

ScreenOutput(tgoto(CM,x-1,y-1),1); }
/* ------------------------------------------------------------------------ */
SetDocCursor()
{ SetTermCursor(CurX-WinOffX,CurY-WinOffY); }
/* ------------------------------------------------------------------------ */
SetPromptCursor()
{ SetTermCursor(PromptCur,WinHeight+1); }
/* ------------------------------------------------------------------------ */
SetInitQuoteCursor()
{ SetTermCursor(QuotePromptLength+1,WinHeight+1); }
/* ------------------------------------------------------------------------ */
SetQuoteCursor()
{ SetTermCursor(QCP+QuotePromptLength+1,WinHeight+1); }
/* ------------------------------------------------------------------------ */
```

```
SetSysCursor()
{ if (ScreenCleared) { SetTermCursor(1,WinHeight+1); } printf("\n"); }
/* ----------------------------------------------------------------------------------- */
ScrollUpAdjust(n)   register int n;
{
 register int y=n;
 if (n!=0) { ErasePromptLine(); ScreenOutput(tgoto(CM,0,WinHeight),1);
             while (y!=0) { ScreenOutput(DD,WinHeight+1+y-n); y--; }
             while (y+n<WinHeight) { WinLineOK[y+1]=WinLineOK[y+n+1]; y++; }
             PromptCur=WinWidth; PromptDisplayed=FALSE;
             ReDisplayPromptLine(); WindowLinesBad(WinHeight-n+1,WinHeight);
           }
}
/* ----------------------------------------------------------------------------------- */
ScrollDownAdjust(n,line)    register int n;   register int line;
{
 register int y=n;
 if (n!=0) { ScreenOutput(tgoto(CM,0,line-1),1);
             while (y!=0) { ScreenOutput(SR,WinHeight-line+1+y-n); y--; }
             while (y+line+n<=WinHeight)
                       { WinLineOK[WinHeight-y]=WinLineOK[WinHeight-y-n]; y++; }
             PromptCur=WinWidth; PromptDisplayed=FALSE;
             ReDisplayPromptLine(); WindowLinesBad(line,line+n-1);
           }
}
/* ----------------------------------------------------------------------------------- */
ReDisplayPromptLine()
{
 char schstring[FNameMax];   char rplstring[FNameMax];
 if (Not(CharAvailable()))
       { PromptCur=WinWidth;
         if (EState==State_Doc)
                  { if      (OP==UpSearch)
                              { PromptMessage(SearchingUpFor); PromptMessage(Found); }
                    else if (OP==DownSearch)
                              { PromptMessage(SearchingDownFor); PromptMessage(Found);}
                    else if (OP==Replace)
                              { CopySBuffToString(schstring);
                                SetPromptMsg(schstring); PromptMessage(Replaced);
                                CopyRBuffToString(rplstring);
                                SetPromptMsg(rplstring); PromptMessage(With);
                              }
                    else if (OP==Input)
                              { PromptMessage(ReadingFile); PromptMessage(Done); }
                    else
                              { PromptMessage(EditingFile); }
                  }
         else
                  { PromptMessage(ShowQuotePrompt); DisplayQuoteBuffer();
                    SetQuoteCursor();
                  }
       }
}
/* ----------------------------------------------------------------------------------- */
NegImage()
{ ScreenOutput(SO,1); }
/* ----------------------------------------------------------------------------------- */
PosImage()
{ ScreenOutput(SE,1); }
/* ----------------------------------------------------------------------------------- */
ClearToEndOfLine()
{ ScreenOutput(CE,1); }
/* ----------------------------------------------------------------------------------- */
RingBell()
```

```
{ ScreenOutput(BL,1); }
/* ------------------------------------------------------------------------------ */
ErasePromptLine()
{ int i=0;          PosImage(); PromptCur=1; SetPromptCursor();
                    while(i++!=WinWidth-1) { printf("%c",' '); }    }
/* ------------------------------------------------------------------------------ */
DrawBlankPromptLine()
{ int i=0;          NegImage(); SetPromptCursor();
                    while(i++!=WinWidth-1) { printf("%c",' '); } PosImage();    }
/* ------------------------------------------------------------------------------ */
CursorToNextLine()
{ ScreenOutput(OD,1); }
/* ------------------------------------------------------------------------------ */
printCSI()                                          /* print CSI sequence for vt220 */
{ printf("%c%c",ESC,'['); }
/* ------------------------------------------------------------------------------ */
```

```
                              /* externglobals.c */
                            /* ================== */

#include "consts.c"
/* ------------------------------------------------------------------------------  */
extern int      LP,RP,CP;
extern char     Arr[Maxx];

extern int      CurX,CurY,StartIn,EndIn,DocNL,WSRem,NLRem;

extern int      WSIns,NLIns,PageWidth,PageHeight;

extern int      MP,PP;
extern char     PArr[Maxx];

extern char     QArr[QMaxx],SArr[QMaxx],RpArr[QMaxx];
extern int      QP,QCP,SP,RpP;
extern int      MatchedLength;
extern char     FName[FNameMaxx];
extern int      FP;
extern int      EState;

extern int      WinOffX,WinOffY;
extern int      WinStartIn;
extern int      WinLineOK[MaxWinHeight+2];
extern int      WinHeight,WinWidth;
extern int      HalfWinHeight,HalfWinWidth;
extern int      QtrWinHeight,QtrWinWidth;

extern char     KbdArr[kbdMaxx];
extern int      Kbdptr;
extern int      HomePath;
extern int      Key;
extern int      OP;
extern int      OPType;
extern char     OPChar;
extern int      OPArr[100];

extern int      NoInterrupt;
```

```
extern int    Backup;

extern int    PromptDisplayed;
extern int    DocChanged;
extern int    PromptCur;
extern char   PromptMsg[FNameMaxx];

extern short  ospeed;
extern int    TermType;
extern struct sgttyb Sgttyb;
extern int    fcntlFlag;
extern int    TerminalSet;
extern int    ScreenCleared;

extern char   PC;
extern char   CM[cntrlsize];
extern char   RC[cntrlsize];
extern char   DD[cntrlsize];
extern char   SD[cntrlsize];
extern char   SE[cntrlsize];
extern char   CE[cntrlsize];
extern char   BL[cntrlsize];
extern char   SR[cntrlsize];
extern char   CL[cntrlsize];
extern char   BC[cntrlsize];
extern char   UP[cntrlsize];

/* ------------------------------------------------------------------------------ */
```

```
                              /* kbdinterpret.c */
                             /* ================ */

#include "externglobals.c"

/* ------------------------------------------------------------------------------ */
KbdRead()
{
 if      (strcmp(TermType,"vt220")==0) { vt220Interpret(); }
 else if (strcmp(TermType,"sun")==0)   { SunInterpret(); }
}
/* ------------------------------------------------------------------------------ */
vt220Interpret()                                      /* keyboard interpret for vt220 */
{
 char x=GetNextChar();
 if      (x>=' ' && x<='~')  { OP=InsertChar; OPChar=x; }
 else if (x==RET)            { OP=InsertChar; OPChar=nl; }
 else if (x==TAB)            { OP=InsertChar; OPChar=x; }
 else if (x==DEL)            { OP=LeftDeleteChar; }
 else if (x==ESC)            { x=GetNextChar();
                               if      (x=='[') { vt220InterpretCSI(); }
                               else if (x=='O') { vt220InterpretSS3(); }
                               else             { OP=NotImplemented; }
                             }
 else                        { OP=NotImplemented; }
}
/* ------------------------------------------------------------------------------ */
vt220InterpretCSI()                                   /* function and editing keys */
```

```
{
 char c;   char d;   char e;   int  i;   int j;
 c=GetNextChar(); d=GetNextChar();
 if (d=='~')                                                          /* editing keys */
     switch (c)
      {
       case '1' : break;                                              /* Find        */
       case '2' : OP=Mark; break;                                     /* Insert Here */
       case '3' : break;                                              /* Remove      */
       case '4' : break;                                              /* Select      */
       case '5' : break;                                              /* Prev Screen */
       case '6' : break;                                              /* Next Screen */
      }
 else {
       e=GetNextChar();
       if (e=='~')                                                    /* function keys */
           {
             i=(10*(c-'0'))+d-'0';
            switch(i)
             {
              case 17 : break;                                        /* F6    */
              case 18 : OP=CursorUpPage; break;                       /* F7    */
              case 19 : OP=MoveToTop; break;                          /* F8    */
              case 20 : OP=MoveToBot; break;                          /* F9    */
              case 21 : OP=CursorDownPage; break;                     /* F10   */

              case 23 : break;                                        /* F11   */
              case 24 : break;                                        /* F12   */
              case 25 : break;                                        /* F13   */
              case 26 : break;                                        /* F14   */

              case 28 : break;                                        /* Help  */
              case 29 : break;                                        /* Do    */

              case 31 : OP=Cut; break;                                /* F17   */
              case 32 : OP=Lift; break;                               /* F18   */
              case 33 : OP=Paste; break;                              /* F19   */
              case 34 : OP=NotImplemented; break;                     /* F20   */
             }
           }
       else  OP=NotImplemented;
      }
}
/* ------------------------------------------------------------------------------------------ */
vt220InterpretSS3()                          /* arrow keys and auxiliary keypad */
{
 switch (GetNextChar())
  {
   case 'A' : OP=CursorUpLine; break;                                 /* up arrow    */
   case 'B' : OP=CursorDownLine; break;                               /* down arrow  */
   case 'C' : OP=CursorRightChar; break;                              /* right arrow */
   case 'D' : OP=CursorLeftChar; break;                               /* left arrow  */

   case 'P' : OP=LeftMoveLine; break;                                 /* PF1 */
   case 'Q' : OP=RightMoveLine; break;                                /* PF2 */
   case 'R' : OP=LeftDeleteLine; break;                               /* PF3 */
   case 'S' : OP=RightDeleteLine; break;                              /* PF4 */

   case 'p' : break;                                                  /* 0 */
   case 'q' : break;                                                  /* 1 */
   case 'r' : break;                                                  /* 2 */
   case 's' : break;                                                  /* 3 */
   case 't' : OP=LeftMoveChar; break;                                 /* 4 */
   case 'u' : OP=RightMoveChar; break;                                /* 5 */
```

```
    case 'v' : OP=LeftDeleteChar; break;                                                        /* 6 */
    case 'w' : OP=LeftMoveWord; break;                                                          /* 7 */
    case 'x' : OP=RightMoveWord; break;                                                         /* 8 */
    case 'y' : OP=LeftDeleteWord; break;                                                        /* 9 */

    case 'l' : OP=RightDeleteChar; break;                                                   /*  ,  */
    case 'm' : OP=RightDeleteWord; break;                                                   /*  -  */
    case 'n' : break;                                                                       /*  .  */
    case 'M' : break;                                                                       /* Enter */
   }
  }
/* -------------------------------------------------------------------------------------- */
SunInterpret()                                                           /* keyboard read for sun */
{
 char x=GetNextChar();
 if      (x>=' ' && x<='~')  { OP=InsertChar; OPChar=x; }
 else if (x==RET)            { OP=InsertChar; OPChar=nl; }
 else if (x==TAB)            { OP=InsertChar; OPChar=x; }
 else if (x==DEL)            { OP=LeftDeleteChar; }
 else if (x==BS)             { OP=LeftMoveChar; }
 else if (x==ESC)            { x=GetNextChar();
                               if      (x=='[') { SunFunctionInterpret(); }
                               else             { OP=NotImplemented; }
                             }
 else                        { OP=NotImplemented; }
}
/* -------------------------------------------------------------------------------------- */
SunFunctionInterpret()
{
 char c=GetNextChar();   char d=GetNextChar();   char e=GetNextChar();
 char f=GetNextChar();   Key=0;
 if (f=='z') { switch((100*(c-'0'))+(10*(d-'0'))+(e-'0')) {
     case 192 : Key=L1; break;     case 234 : Key=L2; break;     case 203 : Key=L3; break;
     case 235 : Key=L4; break;     case 204 : Key=L5; break;     case 236 : Key=L6; break;
     case 205 : Key=L7; break;     case 237 : Key=L8; break;     case 206 : Key=L9; break;
     case 238 : Key=L10; break;    case 202 : Key=F1; break;     case 225 : Key=F2; break;
     case 226 : Key=F3; break;     case 227 : Key=F4; break;     case 228 : Key=F5; break;
     case 229 : Key=F6; break;     case 230 : Key=F7; break;     case 231 : Key=F8; break;
     case 232 : Key=F9; break;     case 208 : Key=R1; break;     case 209 : Key=R2; break;
     case 210 : Key=R3; break;     case 211 : Key=R4; break;     case 212 : Key=R5; break;
     case 213 : Key=R6; break;     case 214 : Key=R7; break;     case 215 : Key=R8; break;
     case 216 : Key=R9; break;     case 217 : Key=R10; break;    case 218 : Key=R11; break;
     case 219 : Key=R12; break;    case 220 : Key=R13; break;    case 221 : Key=R14; break;
     case 222 : Key=R15; break; } }
 OP=OPArr[Key];
}
/* -------------------------------------------------------------------------------------- */
```

```
                                  /* utilities.c */
                                 /* ============= */

/* -------------------------------------------------------------------------------------- */
char GetNextChar()                                    /* unbuffered single character input, */
{                                                     /* returns next key from keyboard     */
  char c=EOF;   int ptr=0;
  if    (Kbdptr!=0) { c=KbdArr[1]; Kbdptr--;
                      while (ptr!=Kbdptr) { KbdArr[ptr+1]=KbdArr[ptr+2]; ptr++; }
```

```
                   }
  else             { while (c==EOF) read(0,&c,1); }
 return(c);
}
/* ------------------------------------------------------------------------------ */
int CharAvailable()
{
 char c=EOF;
 read(0,&c,1); if (c!=EOF && Kbdptr'<kbdMax) { Kbdptr++; KbdArr[Kbdptr]=c; }
 return(Kbdptr!=0);
}
/* ------------------------------------------------------------------------------ */
delay()
{
 int i=1000; while (i>0) { if (CharAvailable()) i=0; else i--; }
}
/* ------------------------------------------------------------------------------ */
int Not(i)                          /* if "i" FALSE, returns TRUE else returns FALSE */
 int i;
{ if   (i==FALSE) return(TRUE); else return(FALSE); }
/* ------------------------------------------------------------------------------ */
int CnvStringToNum(string,ptr)/* "string" is a string, "ptr" a pointer to the value to   */
                              /* be assigned. if each member of the string is an         */
                              /* integer character, the value is assigned and TRUE       */
                              /* returned, else value left unchanged, and FALSE returned */
 char string[]; int  *ptr;
{
 int j=0;

 while (*string!=0 && IsDigitChar(*string)) { j=(10*j)+(*string-'0'); string++; }
 if (*string==0 && j!=0)                 /* end of string reached, so OK if not zero */
       { *ptr=j; return(TRUE); }
 else                                             /* a non-digit character present */
       { return(FALSE); }
}
/* ------------------------------------------------------------------------------ */
int CnvNumToString(num,string)   int num;   char *string;
{
 int ptr=0;   char tmpstring[FNameMaxx];
 while (num!=0 && ptr<FNameMax-2) { tmpstring[ptr]=('0'+(num%10));  ptr++; num=num/10; }
 while (ptr!=0) { ptr--; *string=tmpstring[ptr]; string++; }  *string='\0';
}
/* ------------------------------------------------------------------------------ */
int IsDigitChar(c)                  /* returns TRUE if '0'<=c<='9', else returns FALSE */
 char c;
{ return('0'<=c && c<='9'); }
/* ------------------------------------------------------------------------------ */
int IsAlphaNum(c)               /* returns TRUE if c alpha-numeric, else returns FALSE */
 char c;
{ return(('a'<=c && c<='z') || ('A'<=c && c<='Z') && ('0'<=c || c<='9')); }
/* ------------------------------------------------------------------------------ */
int Lower(c)                                          /* converts ASCII to lower case */
 char c;
{
 if   (c>='A' && c<='Z') return(c+'a'-'A');
 else                    return(c);
}
/* ------------------------------------------------------------------------------ */
```

```
                          /* consts.c */
                        /* ============== */

/* ------------------------------------------------------------------------------------------ */
#define  Maxx              100001
#define  Max               Maxx-1                            /* array[0] not used! */
#define  QMaxx                 65
#define  QMax              QMaxx-1
#define  FNameMaxx            257
#define  FNameMax      FNameMaxx-1
#define  kbdMaxx              101
#define  kbdMax          kbdMaxx-1
#define  tabstop                8

#define  MaxWinWidth          150                            /* window lengths */
#define  MaxWinHeight          60
#define  MinWinWidth            8
#define  MinWinHeight           8
#define  DefaultWinWidth       80
#define  DefaultWinHeight      24

#define  cntrlsize             32                            /* for termcap */
#define  tcsize              1024

#define  NUL               '\000'                            /* special characters */
#define  PAD               '\000'
#define  BEL               '\007'
#define  BS                '\010'
#define  TAB               '\011'
#define  RET               '\012'
#define  LF                '\012'
#define  CR                '\015'
#define  ESC               '\033'
#define  DEL               '\177'
#define  NullPtr             NULL

#define  sp                   ' '
#define  nl                  '\n'

#define  Left                   1                            /* for ConcDoc1 */
#define  Right                  2

#define  CurAbove               1                            /* for ConcDoc6 */
#define  CurBelow               2

#define  State_Doc              0                            /* for ConcDoc8 */
#define  State_Quote            1

#define  TRUE                   1                            /* booleans */
#define  FALSE                  0

#define  OK                     0                            /* report messages */
#define  SystemError            2
#define  NoTERM                 3
#define  UnknownTerminal        4
#define  NoTermcapFile          5
#define  InadeqTermCap          6
#define  TooManyColsTTY         7
#define  TooManyRowsTTY         8
#define  WindowTooShort         9
#define  WindowTooNarrow       10
#define  ReadError             11
#define  WriteError            12
#define  NoReadPermission      13
```

```
#define  CannotOpenFile        14
#define  BadTerminalType       15
#define  NoFilenameGiven       16
#define  BadLineNumber         17
#define  TooManyFilenames      18
#define  BadCommandSyntax      19
#define  NoWritePermission     20
#define  UnknownOption         21
#define  Full                  22
#define  Top                   23
#define  Bot                   24
#define  TopLine               25
#define  LeftEdge              26
#define  NoTextMarked          27
#define  PBuffEmpty            28
#define  TooNearTop            29
#define  OPNotImplemented     30
#define  QuoteError            31
#define  TopQuote              32
#define  BotQuote              33
#define  FullQuote             34
#define  IllegalEditOp         35
#define  IllegalQuoteChar      36
#define  BellRung              37
#define  DocNotChanged         38
#define  ReadingFile           39
#define  WritingFile           40
#define  EditingFile           41
#define  EditingNewFile        42
#define  Done                  43
#define  UpdatingBackup        44
#define  CannotUpdateBackup    45
#define  Suspended             46
#define  ShowQuotePrompt       47
#define  EditAborted           48
#define  CntrlFound            50
#define  Full_Tabs             51
#define  FileNotExist          52
#define  Directory             53
#define  HitKeyToResume        54
#define  NoCommandGiven        55
#define  SearchingUpFor        56
#define  SearchingDownFor      57
#define  Found                 58
#define  NotFound              59
#define  SBuffEmpty            60
#define  Replaced              61
#define  With                  62
#define  NotMatched            63
#define  ShowStats             64
#define  DBuffEmpty            65
#define  NoFredcapFile         66
#define  CannotOpenFredcap     67
#define  Resumed               68
#define  Interrupted           69
#define  NoHOMEset             70
#define  FilenameTooLong       71

#define  NotImplemented         0
#define  MoveToTop              1                                /* editor commands */
#define  MoveToBot              2
#define  LeftMoveChar           3
#define  RightMoveChar          4
#define  LeftMoveWord           5
```

```
#define RightMoveWord           6
#define LeftMoveLine            7
#define RightMoveLine           8
#define LeftDeleteChar          9
#define RightDeleteChar        10
#define LeftDeleteWord         11
#define RightDeleteWord        12
#define LeftDeleteLine         13
#define RightDeleteLine        14
#define CursorLeftChar         15
#define CursorRightChar        16
#define CursorUpLine           17
#define CursorDownLine         18
#define CursorUpPage           19
#define CursorDownPage         20
#define UpSearch               21
#define DownSearch             22
#define Replace                23
#define Mark                   24
#define Cut                    25
#define Lift                   26
#define Paste                  27
#define Quote                  28
#define CentreWindow           29
#define RefreshDisplay         30
#define ShowDocStats           31
#define Abort                  32
#define Save                   33
#define Write                  34
#define Append                 35
#define Quit                   36
#define Input                  37
#define MoveLineNumberOrError  38
#define Escape                 49
#define InsertChar             40

#define LeftInsert              1                                   /* for OPType */
#define RightInsert             2
#define LeftDelete              3
#define RightDelete             4
#define LeftMove                5
#define RightMove               6
#define NoMove                  7

#define L1                      1                            /* sun function keys */
#define L2                      2
#define L3                      3
#define L4                      4
#define L5                      5
#define L6                      6
#define L7                      7
#define L8                      8
#define L9                      9
#define L10                    10
#define F1                     11
#define F2                     12
#define F3                     13
#define F4                     14
#define F5                     15
#define F6                     16
#define F7                     17
#define F8                     18
#define F9                     19
#define R1                     20
```

```
#define  R2                  21
#define  R3                  22
#define  R4                  23
#define  R5                  24
#define  R6                  25
#define  R7                  26
#define  R8                  27
#define  R9                  28
#define  R10                 29
#define  R11                 30
#define  R12                 31
#define  R13                 32
#define  R14                 33
#define  R15                 34


#define  DefaultTermPath                    "/etc/termcap"
#define  DefaultFredPath                        "~/$TERM"
#define  EditorName                         "fred 0.0"

#define  QuotePrompt                        "QUOTE : "
#define  QuotePromptLength                           8
#define  usage "usage: fred [-b] [-l number] [-] filename"
/* ------------------------------------------------------------------------------ */
```

```
                              /* messages.c */
                             /* ============== */

/* ------------------------------------------------------------------------------ */
AbortMessage(rep,ptr)   int rep;   int ptr;
{
 switch(rep)
  {
   case  UnknownTerminal     : printf("\nUnknown terminal type \"%s\"",TermType); break;
   case  NoTermcapFile       : printf("\nCan't find termcap file"); break;
   case  NoHOMEset           : printf("\nEnvironment variable HOME not found"); break;
   case  InadeqTermCap       : printf("\nInadequate terminal capabilities - %s",TermType);
                               printf(" / %s",ptr); break;
   case  TooManyColsTTY      : printf("\nToo many columns in TTY ... columns=%d",ptr);
                               printf(" max columns=%d",MaxWinWidth); break;
   case  TooManyRowsTTY      : printf("\nToo many rows in TTY ... rows=%d",ptr);
                               printf(" max rows=%d",MaxWinHeight); break;
   case  WindowTooShort      : printf("\nWindow too small ... lines=%d",WinHeight);
                               printf(" min lines=%d",MinWinHeight); break;
   case  WindowTooNarrow     : printf("\nWindow too small ... cols=%d",WinWidth);
                               printf(" min cols=%d",MinWinWidth); break;
   case  Full                : printf("\n\"%s\" too big for editor",FName);
                               printf("\n%s capacity:%d",EditorName,Max); break;
   case  Full_Tabs           : printf(" too big for editor \(tabs expanded\)");
                               PosImage(); printf("\n\n%s capacity:%d",EditorName,Max);
                               break;
   case  Directory           : printf("\n\"%s\" is a directory",FName); break;
   case  NoReadPermission    : printf("\nRead permission denied for \"%s\"",FName); break;
   case  CannotOpenFile      : printf("\nCannot open \"%s\"",FName); break;
   case  BadTerminalType     : printf("\nfred: not available for ");
                               printf("terminal type \"%s\"",FName); break;
   case  FilenameTooLong     : printf("\nFilename \"%s\" too long",ptr); break;
```

```
    case NoFilenameGiven     : printf("\nfred: no filename given ... "); printf(usage); break;
    case BadLineNumber       : printf("\nfred: bad line number \"%s\"... ",ptr);
                               printf(usage); break;
    case TooManyFilenames    : printf("\nfred: too many filenames ... "); printf(usage);
                               break;
    case BadCommandSyntax    : printf("\nfred: bad syntax ... "); printf(usage); break;
    case NoWritePermission   : printf("\nWrite permission denied for file \"%s\"",FName);
                               break;
    case UnknownOption       : printf("\nfred: Unknown option %s ... ",ptr);
                               printf(usage); break;
    case NoFredcapFile       : printf("\nCan't find fredcap file"); break;
    case CannotOpenFredcap   : printf("\nCan't open fredcap file \"%s\"",FName); break;
  }
}
/* ------------------------------------------------------------------------------------ */
PromptMessage(rep)   int rep;
{
 NegImage(); switch(rep)
 {
    case Top :
         Prompt(1,"At top of document"); MsgEnd(); break;
    case Bot :
         Prompt(1,"At bottom of document"); MsgEnd(); break;
    case Full :
         Prompt(1,"Editor capacity reached"); MsgEnd(); break;
    case Full_Tabs :
         Prompt(0,"Editor capacity exceeded expanding tabs"); MsgEnd(); break;
    case TopLine :
         Prompt(1,"At top line of document"); MsgEnd(); break;
    case LeftEdge :
         Prompt(1,"At left edge of document"); MsgEnd(); break;
    case NoTextMarked :
         Prompt(1,"No Text Marked"); MsgEnd(); break;
    case PBuffEmpty :
         Prompt(1,"Paste buffer empty"); MsgEnd(); break;
    case TooNearTop :
         Prompt(1,"Too near top of document"); MsgEnd(); break;
    case DocNotChanged :
         Prompt(1,"\""); Prompt(0,FName); Prompt(0,"\"");
         Prompt(0," Not changed ... Not written"); MsgEnd(); break;
    case QuoteError :
         Prompt(1,"QUOTE ; abort  a  i  q  s  w  !cmd  number"); RingBell();
         MsgEnd(); break;
    case ReadError :
         Prompt(0,"Read error"); MsgEnd(); break;
    case WriteError :
         Prompt(0,"Write error"); MsgEnd(); break;
    case NoCommandGiven :
         Prompt(1,"No command given"); MsgEnd(); break;
    case OPNotImplemented :
         Prompt(1,"Not implemented"); RingBell(); MsgEnd(); break;
    case SBuffEmpty :
         Prompt(1,"Search buffer empty"); MsgEnd(); break;
    case ReadingFile :
         Prompt(1,"Reading "); Prompt(0,"\""); Prompt(0,PromptMsg); Prompt(0,"\"");
         Prompt(0," ... "); break;
    case WritingFile :
         if   (Backup) { Prompt(0,"Writing "); }
         else          { Prompt(1,"Writing "); }
         Prompt(0,"\""); Prompt(0,PromptMsg); Prompt(0,"\""); Prompt(0," ... ");
         break;
    case EditingFile :
         Prompt(1,EditorName); Prompt(0," Editing "); Prompt(0,"\"");
         Prompt(0,FName); Prompt(0,"\"");
```

```
            if (DocChanged) { Prompt(0," *"); };
            PromptDisplayed=TRUE; PosImage(); break;
    case EditingNewFile :
            Prompt(1,EditorName); Prompt(0," Editing "); Prompt(0,"\"");
            Prompt(0,FName); Prompt(0,"\""); Prompt(0," (new file)");
            PromptDisplayed=TRUE; PosImage(); break;
    case Done :
            Prompt(0,"Done"); MsgEnd(); break;
    case UpdatingBackup :
            Prompt(1,"Updating "); Prompt(0,"\""); Prompt(0,PromptMsg); Prompt(0,"\"");
            Prompt(0," ... "); break;
    case CannotUpdateBackup :
            Prompt(1,"Can't update "); Prompt(0,"\""); Prompt(0,PromptMsg);
            Prompt(0,"\""); Prompt(0," ... "); RingBell(); break;
    case NoReadPermission :
            Prompt(1,"Read permission denied for "); Prompt(0,"\"");
            Prompt(0,PromptMsg); Prompt(0,"\""); MsgEnd(); break;
    case Directory :
            Prompt(1,"\""); Prompt(0,PromptMsg); Prompt(0,"\"");
            Prompt(0," is a directory"); MsgEnd(); break;
    case FileNotExist :
            Prompt(1,"\""); Prompt(0,PromptMsg); Prompt(0,"\"");
            Prompt(0," does not exist"); MsgEnd(); break;
    case CannotOpenFile :
            Prompt(1,"Can't open "); Prompt(0,"\""); Prompt(0,PromptMsg);
            Prompt(0,"\""); MsgEnd(); break;
    case ShowQuotePrompt :
            Prompt(1,QuotePrompt); break;
    case EditAborted :
            Prompt(1,"Edit aborted ... \""); Prompt(0,FName);
            Prompt(0,"\" Not written"); break;
    case CntrlFound :
            Prompt(0," (Control chars discarded)"); RingBell(); MsgEnd(); break;
    case HitKeyToResume :
            PosImage(); printf("\nHit <Return> to resume editing \"%s\"... ",FName); break;
    case SearchingUpFor :
            Prompt(1,"Searching (up) for \""); Prompt(0,PromptMsg);
            Prompt(0,"\" ... "); break;
    case SearchingDownFor :
            Prompt(1,"Searching (down) for \""); Prompt(0,PromptMsg);
            Prompt(0,"\" ... "); break;
    case NotFound :
            Prompt(0,"Not found"); MsgEnd(); break;
    case Found :
            Prompt(0,"Found"); MsgEnd(); break;
    case Replaced :
            Prompt(1,"Replaced \""); Prompt(0,PromptMsg); Prompt(0,"\""); break;
    case With :
            Prompt(0," With \""); Prompt(0,PromptMsg); Prompt(0,"\""); MsgEnd(); break;
    case NotMatched :
            Prompt(1,"Not matched with \""); Prompt(0,PromptMsg); Prompt(0,"\"");
            MsgEnd(); break;
    case ShowState:
            Prompt(1,"Chars: "); GetAsString(Max+LP-RP); Prompt(0,PromptMsg);
            Prompt(0," \("); GetAsString(CP+1); Prompt(0,PromptMsg); Prompt(0,"\)");
            Prompt(0,"  Lines: "); GetAsString(DocNL+1); Prompt(0,PromptMsg); Prompt(0," \(");
            GetAsString(CurY); Prompt(0,PromptMsg); Prompt(0,"\)"); MsgEnd(); break;
    case DBuffEmpty :
            Prompt(1,"Delete buffer empty"); MsgEnd(); break;
    case Interrupted :
            Prompt(0," \(Interrupted\)"); RingBell(); MsgEnd(); break;
    case Resumed :
            Prompt(0," \(Resumed\)"); MsgEnd(); break;
}
```

```
 fflush(stdout);
}
/* ---------------------------------------------------------------------------------- */
Prompt(curposn,msg)   int curposn;   char msg[];
{
 int prevPromptCur=PromptCur;   int ptr=0;   int count;
 if   (curposn==1) { PromptCur=1; }
 SetPromptCursor();
 while (PromptCur<WinWidth && msg[ptr]!='\0') { printf("%c",msg[ptr]); ptr++; PromptCur++; }
 count=PromptCur;
 while (count<prevPromptCur) { printf("%c",sp); count++; }
 SetPromptCursor();
}
/* ---------------------------------------------------------------------------------- */
MsgEnd()
{
 /* fflush(stdout); */ PromptDisplayed=FALSE;  PosImage();
}
/* ---------------------------------------------------------------------------------- */
SetPromptMsg(string)   char *string;
{
 strcpy(PromptMsg,string);
}
/* ---------------------------------------------------------------------------------- */
int GetAsString(num)   int num;
{
 char string[FNameMaxx];    CnvNumToString(num,string); SetPromptMsg(string);
}
/* ---------------------------------------------------------------------------------- */
```