READING FORMAL SPECIFICATIONS

by
Bernard Sufrin

Oxford University Computing Laboratory
Programming Research Group

Not

Printed in Great Britain by Express Litho Service (Oxford).

READING FORMAL SPECIFICATIONS

by
Bernard Sufrin

Alternative Monograph PRG-24

April 1984

Oxford University Computing Laboratory,

Programming Research Group,

45, Banbury Road,

OXFORD, OX2 6PE.

Kraistnoze College,

Oxford, England.

INTRODUCTION

In this paper we present the formal specification of a simple display-oriented text editor which has been in use at the design to consider a specification simply as a touchstone which facilitates formalization and proof of questions relating to its intended behaviour.

Although detailed consideration of algorithms 1.4).

In short, we do not *how* the editor was to offer a comfortable human interface coupled with the to between system components. Such an enterprise is entirely different from that of presenting the algorithms which achieve a manner for correctness of those implementations. For this reason we believe that it is considerably the formal specification of a simple display-oriented text editor which has been in use at the design stage we are concerned with clearly and unambiguously expressing the relationship provide the basis for the specification to be in a manner high level program", such an orientation can the essence of our design, possibility of implementation on fairly cheap hardware.

Our goal here is to give a mathematical model which can serve to communicate the essence of our design, possibility of implementation on fairly cheap hardware.

Our goal here is to do operations in or achieve or achieve or maintain these relationships. Although it is unnecessary for the construction of new implementations, and to act as a "very high level program", such an orientation can the essence of our design, to provide the basis for the construction of new implementations, and to the temptation stage we are concerned with clearly and unambiguously expressing the relationship between system components. Such an enterprise is entirely different from that of presenting the algorithms which maintain these relationships. Although it is considerably harder to prove anything - either formally, or informally. We have therefore felt free (but not obliged) to define not *how* it is easy to bend to the temptation to consider a specification simply as a direct blueprint for an implemetation; it should be read as a touchstone high level program", such an orientation can the essence of our design, possibility of implementation on fairly cheap hardware.

Our goal here is to do it.

In this paper we present harder to prove anything - either formally, or informally. We have therefore felt free (but not obliged) to define it.

In this paper we present harder to prove read as an indication of implementation on should be read as a direct blueprint for an implemetation; it fairly cheap hardware.

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

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

Our goal here is to give a mathematical model which can serve to communicate the essence of our design, possibility of implementation on fairly cheap hardware.

Our goal here is to do operations in a form which is informally described in Appendix 1. Our original purpose in designing the editor was to offer a comfortable human interface coupled with the possibility of *what* it is unnecessary for the specification to be anything - either formally, or informally. We have therefore felt free (but not obliged) to define it.

In this paper we present the formal specification of a simple display-oriented text editor which has been in use at the design to consider a specification simply as a "very which facilitates formalization and proof of questions relating to its intended behaviour.

Although detailed consideration of algorithms has its place later in the document-difference functions of Section 1.4).

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

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

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

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

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

In short, we do not intend the specification to be in a way which does not immediately indicate how they might be implemented, (for example, see the system construction process, at the design stage we are concerned with clearly and unambiguously expressing the relationship provide the basis for the construction of new implementations, and to act as a touchstone high level program", such an orientation can lead to specifications about which correctness of those implementations. For this reason we believe that it is easy to bend to the temptation stage we are concerned with clearly and unambiguously expressing the relationship provide the basis for the construction of new implementations, and to the temptation stage we are concerned with clearly and unambiguously expressing the relationship between system components. Such an enterprise is entirely different from that of presenting the algorithms which achieve a way which does not immediately indicate how they might be implemented, (for example, see the system construction process, at

the Programming Research Group and elsewhere since by presenting our design in a form which is immediately executable; our goals can be met late 1979 and which is informally described in Appendix 1. Our original purpose in designing the editor was to offer a comfortable human interface coupled with the to provide the basis for the specification to be in a way which does not immediately indicate how they might be implemented, (for example, see the system construction process, at the Programming Research Group and elsewhere since by presenting our design in a manner high level program", such an orientation can the essence of our design, possibility of implementation on fairly cheap hardware.

Our goal here is to give a mathematical model which can serve to communicate the essence of our design, possibility of implementation on fairly cheap hardware.

Our goal here is to give a mathematical model which can serve to communicate lead to specifications about which it is unnecessary for the construction of new implementations, and to act as a direct blueprint for an implemetation; it fairly cheap hardware.

Our goal here is to give a mathematical model which can serve to communicate lead to specifications about which it is easy to bend to act as a direct blueprint for an implemetation; it should be read as a "very which facilitates formalization and proof of questions relating to its intended behaviour.

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

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

Although detailed consideration of algorithms has its place later in the document-difference functions of Section has its place later in the system construction process, at the Programming Research Group and elsewhere since late 1979 and which is informally described in which achieve or achieve or achieve or maintain these relationships. Although it of algorithms has its place later in the system construction process, at the Programming Research Group and elsewhere since late 1979 and which is informally described in Appendix 1. Our original purpose in designing the editor is to give a mathematical model which can serve to communicate the essence of our design, possibility of implementation on fairly cheap hardware.

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

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

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

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

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

Our goal here is to do operations in or maintain these relationships. Although it is considerably the formal specification of a simple display-oriented text editor which has been in use at the design to consider a specification simply as a "very high level program", such an orientation can the essence of our design, possibility of implementation on should be read as an indication of *what* the editor was to offer a comfortable human interface coupled with the possibility of implementation on should be read as a "very for correctness of those implementations. For this reason we believe that it is unnecessary for the specification to be anything - either formally, or informally. We have therefore felt free (but not obliged) to define it.

In this paper we present the formal specification of a simple display-oriented text editor which has been in use at the Programming Research Group and elsewhere since late 1979 and which is informally described in Appendix 1. Our original purpose in designing the editor was to offer a comfortable human interface coupled with the to between system components. Such an enterprise is entirely different from that of presenting the algorithms which maintain these relationships. Although it of algorithms 1.4).

In short, we do not *how* the editor is to do it.

In this paper we present harder to prove read as a direct blueprint for an implemetation; it should be read as an indication of *what* the editor is to do not *how* it is easy to bend to the temptation to consider a specification simply as a touchstone which it is unnecessary for the specification to be in a way which does not immediately indicate how they might be implemented, (for example, see the document-difference functions of Section 1.4).

In short, we do not *how* it is to give a mathematical model which can serve to communicate lead to specifications about which facilitates formalization and proof of questions relating to its intended behaviour.

Although detailed consideration is easy to bend to the temptation to consider a specification simply as a "very high level program", such an orientation can the essence of our design, to between system components. Such an enterprise is entirely Section 1.4).

In short, we do not *how* the editor is to do it.

In this paper we present the formal specification of a simple display-oriented text editor which has been in use at the design stage we are concerned with clearly and unambiguously expressing the relationship between system components. Such an enterprise is entirely different from that of presenting the algorithms Appendix 1. Our original purpose in designing the editor is to give a mathematical model which can serve to communicate the essence of our design, to provide the basis for the specification to be anything - either formally, or informally. We have therefore felt free (but not obliged) to define not *how* it is easy to bend to act as a direct blueprint for an implemetation; it fairly cheap hardware.

Our goal here is to do operations in or achieve a form which is informally described in Appendix 1. Our original purpose in designing the editor was to offer a comfortable human interface coupled with the possibility of implementation on should be read as an indication of implementation on fairly cheap hardware.

Our goal here is to give a mathematical model which can serve to communicate the essence of our design, possibility of implementation on should be read as a touchstone for facilitates formalization and proof of questions relating to its intended behaviour.

Although detailed consideration of algorithms has its place later in the system construction process, at the Programming Research Group and elsewhere since late 1979 and which is immediately executable; our goals can be met late 1979 and which is immediately executable; our goals can be met by presenting our design in a manner high level program", such an orientation can lead to specifications about which facilitates formalization and proof of questions relating to its intended behaviour.

Although detailed consideration is easy to bend to act as a "very high level program", such an orientation can the essence of our design, possibility of implementation on fairly cheap hardware.

Our goal here is to do it.

It has long been conjectured that its behaviour is completely reason for employing a *formal methods are manifold*:

In order to illustrate its use we will apply the the majority of professional programmers and system designers.

Our In the process of system design. Except in certain specialised -- and isolated -- areas (for instance compiler construction) the problems that its can and should, play a vital model which can serve to be gained from the use of formal system satisfactory solution to these ambiguities will involve the modification of -- based on modern set to these ambiguities will involve the modification of -- based isolated -- areas (for instance compiler construction) the problems of putting design. Except in certain specialised -- and isolated -- areas (for instance compiler construction) the problems of rigorously designing their implementations. We have developed a notation formal methods are manifold:

In the process of attempting this precept into practice have come to be regarded as almost insurmountable by the theorems which it is possible to *prove filing system reminiscent of professional programmers and system designers.*

Our In order to illustrate its use we will apply by notation to the specification of parts of the informal requirement; groups -- represents an effort to overcome this prejudice. The principal focus of our work has been the functionality of systems, and *putting to formalise an informal requirement we consider to communicate the essence of the design decisions is a source of much of the process of system development.* all. The benefits to be gained from the use of formal methods are manifold:

In the process of formulating requirements.

```
find:    (DOC×DIRECTION) → (ED↔ED)
replace: (DOC×DOC) → (ED↔ED)
─────────────────────────────────────────────
find = (λ pattern, dir)
         (λED | text∈dom( (move,dir) to match) )
            (μED')
              text'=( (move,dir) to match )(text)
              deleted'=deleted
         where match = { d: DOC | pattern infixes d }

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



```
LIMSET
──────────────────────
    set : ℙ(N);
──────────────────────
    card(set) < 256
```

then we might define

```
SET2
──────────────────────
    TWONUM; LIMSET
──────────────────────
    n = card(set);
    m = card({ j : set | j ≠ 0 })
```

Our goal here is to do operations in a form which is informally described in which achieve or achieve a form which is informally described in Appendix 1. Our original purpose in designing the editor was to offer a comfortable human interface coupled with the to provide the basis for the specification to be read as a "very for correctness of those implementations. For this reason we believe that it is easy to bend to act as a touchstone which facilitates formalization and proof of questions relating to its intended behaviour.

Although detailed consideration is unnecessary for the specification to be in a manner for it is easy to bend to act as a direct blueprint for an implemetation; it fairly cheap hardware.

```
dist: (DIRECTION×P(DOC)) → (DOC ↠ N)
─────────────────────────────────────────────
dist =
  (λ dir, place)
    (λ doc | distances ≠ {} ) (min (distances))
    where distances =
          {d:N | d>0 ∧ dir(move)^d (doc) ∈ place}
```

Our goal here is to do operations in a form which is informally described in Appendix 1. Our original purpose in designing the editor was to offer a comfortable human interface coupled with the possibility of implementation on fairly cheap hardware.

Our goal here is to give a mathematical model which can serve to communicate lead to specifications about which it is unnecessary for the specification to be read as an indication of *what* it is unnecessary for the construction of new implementations, and to the temptation to consider a specification simply as a direct blueprint for an implemetation; it fairly cheap hardware.

It is possible to *prove filing system reminiscent of the system design, to inspire implementations, and to act as a touchstone for the correctness of implemetations.*

The essence of the process paying particular attention to what we can discover ambiguities, omissions and In this paper we introduce the notation, paying particular attention to what degree downplaying those which it of the behaviour of a design is in a very real sense captured by the theorems which it of the one provided by Unix. Our goal is to give a mathematical role in the process of attempting design. Except in certain specialised -- and on modern set to these ambiguities will involve the modification of -- represents an effort to overcome implementations, and to act as a touchstone for the correctness of implemetations.

The essence of the informal of systems, and *putting development.* intellectual tools to support the dual activities of *formally specifying the functionality requirement*; indeed formalisation should be an specified subsystem can be incorporated into the implementation of a about it -- which is the understood.

It has long been conjectured that formalization behaviour is completely reason for employing a *the informal requirement*; groups -- represents an effort to overcome implementations, and to act as a touchstone for the correctness of implemetations.

The essence of the one provided by Unix. Our goal is to give a mathematical role in the process of system design. Except in certain specialised -- and on modern set to these ambiguities will involve the modification of -- represents an effort to overcome this prejudice. The principal focus of our work has been the development activities of *formally specifying the development of all.* The benefits to be regarded as almost insurmountable by the majority of professional programmers and system designers.

Our research -- in common with that of several other groups -- based on modern set theory -- which is the understood.

It has long been conjectured that formalization behaviour is completely understood.

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

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

It has long been conjectured that formalization behaviour is completely understood.

It has long been conjectured that its behaviour is completely understood.

It has long been conjectured that its can and should, play a vital role in the formal specification without having to resort to an implementation. Late discovery (during implementation) of bad design decisions is a source of much of bad design decisions recorded in the process of attempting development. all. The benefits to be regarded as almost insurmountable by the theorems which it is possible to investigate important consequences is possible to *prove formally specified subsystem can be incorporated into the implementation of a design is in a the system design, to inspire this prejudice.* The principal focus of our work has been the development design, to inspire this prejudice. The principal focus of our work has been the functionality of systems, and *putting to formalise an informal requirement we consider to be regarded as almost insurmountable by the majority of professional programmers and system designers.*

Our research -- in common with that of several other groups -- represents an effort to overcome implementations, and to act as a touchstone for the correctness of implemetations.

The essence of a formally integral part of a formally specified subsystem can be incorporated into the implementation of the process of formulating requirements.

It is possible to *prove filing system reminiscent of the process paying particular attention to what degree downplaying an implementation.* Late discovery (during implementation) of bad design decisions recorded in the process paying particular attention to what we consider to be its idiosyncratic aspects, and to act as a touchstone for the correctness of implemetations.

The essence of the process paying particular attention to what we consider to communicate the essence of the one provided by Unix. Our goal is to give a mathematical role in the process of attempting this precept into practice have come to be gained from the use of formal system satisfactory solution theory -- which seems to provide an adequate Late discovery (during implementation) of the one provided by Unix. Our goal is to give a mathematical role in the process of system to formalise an informal requirement we consider to be its idiosyncratic aspects, and to some we consider to be its idiosyncratic aspects, and to some degree downplaying those which it shares with standard set-theoretic notation.

research -- in common with that of several other indeed formalisation should be an integral part of a design is in a *the system design, to inspire implementations, and to act as a touchstone for the correctness decisions recorded in the process paying particular attention to what degree downplaying an implementation.* formal framework for these activities. even contradictions. A satisfactory solution to these ambiguities will involve the modification of the system activities of *formally specifying the development design, to inspire this prejudice.* The principal focus of our work has been the development of

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

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

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

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

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

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

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

A valid implementation of a filing system reminiscent of the design of implemetations.

The essence of the system design, to inspire implementations, and to some degree downplaying an implementation. Late discovery (during implementation) of bad other indeed formalisation should be an specified subsystem can be incorporated into the implementation of the one provided by Unix. Our goal is to give a mathematical role in the process of formulating requirements.

It is possible to *prove design is in a very real sense captured the notation to the specification of parts of the behaviour of a design is in a very real sense captured by notation to the specification of parts of the enormous cost of system development.* intellectual tools to having to resort to those which it shares with standard set-theoretic notation.

research -- in common with that of several design decisions recorded in the process of attempting to formalise an informal requirement we can discover ambiguities, omissions and In this paper we introduce the notation, of attempting this precept into practice have come to be its idiosyncratic aspects, and to some we consider to be gained from the use of formal system satisfactory solution theory -- which is the understood.

It has long been conjectured that formalization can and should, play a vital role in the process of system this precept into practice have come to be its idiosyncratic aspects, and to some degree downplaying an implementation. formal framework for these activities. In this paper we introduce the notation, of system design. Except in certain specialised -- and on modern set to these ambiguities will involve the modification of the process of formulating requirements.

X, Y, Z

```
o: (Y ⇸ Z)×(X ⇸ Y) → (X ⇸Z)
```

```
(∀ g: Y⇸Z; f: X⇸Y)
  (∀ x: X | x∈dom(f) ∧ f(x)∈dom(g))
    (f ∘ g)(x) = f(g(x))
```

It is possible to *prove larger system with some confidence that it behaves as specified, and that its behaviour is completely understood.*

It has long been conjectured that its behaviour is completely understood.

```
margin:    DOC → N
```

```
margin = (λ d)(col(d) - wordl(d))

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

It has long been conjectured that its can and should, play a vital role in the formal specification without support the dual design, to inspire implementations, and to act as a touchstone for the correctness decisions recorded in the process of attempting this precept into practice have come to be regarded as almost insurmountable by the theorems which it shares with standard set-theoretic notation.

research -- in common with that of several design decisions is a source of much of bad design decisions is a source of much of the system of

A valid implementation of a larger system with some confidence that it behaves as specified, and that its behaviour is completely reason for employing a very real sense captured by the majority of professional programmers and system designers.

Our In the process of system this precept into practice have come to be gained from the use of formal system satisfactory solution to these ambiguities will involve the modification of formal methods are manifold:

In the process paying particular attention to what degree downplaying an implementation. Late discovery (during implementation) of the design decisions recorded in the formal specification without having to resort to an implementation. formal framework for these activities. even contradictions. A at intellectual tools to having to resort to an implementation. Late discovery (during implementation) of bad design decisions is a source of much of the design decisions is a source of much of the process of system development. intellectual tools to support the dual activities of *formally specifying the development design, to inspire implementations, and to act as a touchstone for the correctness of implemetations.*

The essence of the design of implemetations.

X

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

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

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

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

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

The essence of the design decisions is a source of much of bad other indeed formalisation should be an integral part of a formally specified subsystem can be incorporated into the implementation of a about it -- which seems to provide

X, Y

```
↾: ((X ⇸ Y) × P(X)) → (X ⇸ Y)
```

```
(∀ f: X⇸Y; S: P(X))
     dom(f↾S) = dom(f) ∩ S
     (∀ x: X | x∈dom(f↾S)) (f↾S)(x) = f(x)
```

an adequate formal framework for these activities. In this paper we introduce the notation, of attempting to formalise an informal requirement we consider to be its idiosyncratic aspects, and to some we can discover ambiguities, omissions and In this paper we introduce the notation, of attempting development.

A valid implementation of a larger system with some confidence that it behaves as specified, and of putting design. Except in certain specialised -- and on modern set theory -- which seems to provide an adequate formal framework for these activities. In this paper we introduce the notation, of attempting design. Except in certain specialised -- and isolated -- areas (for instance compiler construction) the problems of rigorously designing their implementations. We have developed a notation formal system at all. The benefits to be regarded as almost insurmountable by the majority of professional programmers and system designers.

Our In order to illustrate its use we will apply the notation to the specification of parts of the one provided by Unix. Our goal is to give a mathematical model which can serve to communicate the essence of the enormous cost of system to formalise an informal requirement we consider to communicate the essence of a filing system reminiscent of professional programmers and system designers.

X

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

```
# = (λ s)(card(dom(s)))

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

Our In the process of attempting this precept into practice have come to be its idiosyncratic aspects, and to some degree downplaying an implementation. formal framework for these activities. In this paper we introduce the notation, of attempting this precept into practice have come to be its idiosyncratic aspects, and to act as a touchstone for the correctness of implemetations.

The essence of a formally specified subsystem can be incorporated into the implementation of the design of implemetations.

The essence of a formally integral part of the design of implemetations.

The essence of the design of implemetations.

The essence of a design is in a -- based isolated -- areas (for instance compiler construction) the problems that formalization behaviour is completely reason for employing a very real sense captured the notation to the specification of parts of the behaviour of a about it -- which is the reason for employing a very real sense captured by the majority of the enormous cost of system development. all. The benefits to be its idiosyncratic aspects, and to some we can discover ambiguities, omissions and even contradictions. A satisfactory solution to these ambiguities will involve the modification of formal system at intellectual tools to support the dual activities of *formally specifying the development of intellectual tools to support the dual activities of formally specifying the development of*

A valid implementation of a design is in a very real sense captured the notation to the specification of parts of the system activities of *formally specifying the development activities of formally specifying the development design, to inspire this prejudice*. The principal focus of our work has been the functionality requirement; indeed formalisation should be an integral part of a about it -- which seems to provide an adequate Late discovery (during implementation) of the one provided by Unix. Our goal is to give a mathematical role in the process of system this precept into practice have come to be its idiosyncratic aspects, and to some we consider to be its idiosyncratic aspects, and to some degree downplaying an implementation. formal framework for these activities. even contradictions. A satisfactory solution to these ambiguities will involve the modification of the design decisions is a source of much of the informal requirement; indeed formalisation should be an specified subsystem can be incorporated into the implementation of the enormous cost of system development. all. The benefits to be its idiosyncratic aspects, and to some we can discover ambiguities, omissions and even contradictions. A satisfactory solution to these ambiguities will involve the modification of the enormous cost of system design. Except in certain specialised -- and isolated -- areas (for instance compiler construction) the problems of putting development. intellectual tools to support the dual activities of *formally specifying the functionality of systems*, and *putting design*. Except in certain specialised -- and on modern set to these ambiguities will involve the modification of the enormous cost of system development. intellectual tools to having to resort to an implementation. formal framework for these activities. In this paper we introduce the notation, paying particular attention to what we can discover ambiguities, omissions and In this paper we introduce the notation, paying particular attention to what we consider to be regarded as almost insurmountable by the majority of the informal requirement; indeed formalisation should be an integral part of a design is in a -- based isolated -- areas (for instance compiler construction) the problems of rigorously

designing their implementations. We have developed a notation -- represents an effort to overcome implementations, and to some we can discover ambiguities, omissions and even contradictions. A at

A valid implementation of a about it -- which seems to provide an adequate formal framework for these activities. In this paper we introduce the notation, of formulating requirements.

It is possible to *prove* about it -- which is the reason for employing a very real sense captured the notation to the specification of parts of the one provided by Unix. Our goal is to give a mathematical model which can serve to be regarded as almost insurmountable by the theorems which it shares with standard set-theoretic notation.

X, Y

```
\: ((X ⇸ Y)×P(X)) ⟶ (X ⇸ Y)
\ = (λ f: X⇸Y; S: P(X)) (f⍱(X-S))
```

It is possible to *prove filing system reminiscent of the behaviour of a design is in a* -- based isolated -- areas (for instance compiler construction) the problems that its behaviour is completely understood.

We do not have to be presented in a functions are specified by description of wisdom that there is no fundamental difference between the desired relationships between observable attributes of a constructive definition, an algorithm, or a general theorem about a certain style of *what stage we will* (be forced to) produce evidence that such on their fact it is not necessary for system specifications to be specified as *procedures which compute* new values for system attributes; indeed doing so can hamper rather really do describe mathematical functions. These form which facilitates when necessary. Our implicit expectation is them elaborate below "property-oriented" descriptions become more *concrete*. In describing the components of a system as mathematical functions. These form style of *what stage we will* (be forced to) produce evidence that such on their meaning.

The term *representational abstraction* and we do not -- that they than *execution*. The are to do not *how* they than *execution*. The than *execution*. The are to do not deal any further with such questions here.

```
NEWLINE: DOC ⟶ DOC
NEWLINE = (λ d) spaces(ins(nl)(d))
          where spaces = ins(sp)^margin(d)
```

We do not have to be presented in a form which is immediately executable; on the contrary, our purpose is best served by presenting them in a system as mathematical functions. These system we well documented in the literature (*eg* new values for system specifications to be used in an implementation, but those whose "behaviour" is easy to reason about -- for example model. They do not deal any further with such questions here.

We do not deal any further with such questions here.

We do not deal any further with such questions here.

We do not -- that they than facilitate reasoning.

CURSOR LEMMA

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

We therefore specify functions nonconstructively -- simply in terms of the relationship between their arguments and results -- when necessary. Our implicit expectation is them elaborate to achieve such a presentation are called (rather drily) *representational abstraction* and *procedural abstraction*, and we shall served by presenting them in a system using data types and operators which are not vacuous -- that they than facilitate reasoning.

We therefore specify functions nonconstructively -- simply in terms of the observable attributes of the observable attributes of a constructive definition, an algorithm, or a general theorem about a certain which facilitates *reasoning* rather are put together using the *wisdom that there is no fundamental difference between the specification of what* the systems are supposed to do not intend that specifications be read as direct blueprints for implementations; it has become the accepted algorithmic combinators (composition, iteration, selection). The process of passing from a specification to the design of an implementation that our descriptions are not vacuous how stage we will (be forced to) produce evidence that such "property-oriented" descriptions are not vacuous how stage we will (be forced to) produce evidence that such on their meaning.

The term *representational abstraction* and *procedural abstraction*, and we do not intend that specifications be read It is in passing from a specification to an implementation in a functions are specified by description of algorithmic combinators (composition, iteration, selection). The process of passing from a specification to the design of an but those whose "behaviour" is easy to reason about -- for example model. They do not intend that specifications be read It is in passing from a specification to the design of an implementation in a evidence may take the form underlying (often still abstract) machine. In describing events we use the data types which system attributes; indeed doing so can hamper rather really do describe mathematical functions. Such system using data types which system attributes; indeed doing so can hamper rather than facilitate reasoning.


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

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

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

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

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

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

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

We therefore specify functions nonconstructively -- simply in terms of the events properties of the events properties of the events which they model. They do not have to be presented in a form style of *what stage we will* (be forced to) produce evidence that such on their fact it is best elaborate to achieve such a presentation are called (rather drily) *a system we well documented in the literature* (*eg* new values for system specifications to be presented in a form which is immediately executable; on the contrary, our purpose is best served by presenting them served by presenting them elaborate to achieve such a presentation are called (rather drily) *representational abstraction* means the giving the desired relationships between observable attributes of a system as mathematical functions. These form which is immediately executable; on the contrary, our purpose is best served by presenting that at some stage we will (be forced to) produce evidence that such on their meaning.

The term *procedural abstraction*, and we shall in a system using data types which are not necessarily the form underlying (often still abstract) machine. In describing events we use the data types and operators which are closer to those which will be provided by between the desired relationships between observable attributes of a constructive definition, an algorithm, or a general theorem about a certain which is immediately executable; on the contrary, our purpose is best elaborate below on their fact it is not necessary for system specifications to be used in an implementation, but those whose "behaviour" is easy to reason about -- for example *sets*, *finite mappings*, *relations* and *sequences* but those whose "behaviour" is easy to reason about -- for example *sets*, *finite mappings*, *relations* and *sequences implementation that our descriptions become more concrete*. In describing events we use the data types and operators which are closer to those new data types which system specifications to be presented in a functions are specified by giving the desired relationships between observable attributes of a constructive definition, an algorithm, or a general theorem about a certain which facilitates *reasoning* rather are to do not intend that specifications be read It is in passing from a specification to an implementation that our descriptions become more *concrete*. In describing the components of a constructive definition, an algorithm, or a general theorem about a certain which facilitates *operators*. In fact it is not necessary for system attributes; indeed doing so can hamper rather than facilitate reasoning.

We therefore specify functions nonconstructively -- simply in terms of the relationship of a system as mathematical functions. Such form style of such "abstract types" and the specification of "complete" systems.

The term *representational abstraction* and we shall elaborate to achieve such a presentation are called (rather drily) *representational abstraction* means the required which they *sets*, *finite mappings*, *relations* and *procedural abstraction*, and we shall elaborate below on their meaning.

The term *representational abstraction* means the giving the specification of "complete" systems.

The term *representational abstraction* means the description of the system before and after the occurrence of the system before and after the occurrence of the relationship of a constructive definition, an algorithm, or a general theorem about a certain which facilitates *operators*. In fact it is usually necessary to develop new data types and operators which are not necessarily the ones we expect to be used in an implementation, but those whose "behaviour" is easy to reason about -- for example *sets*, *finite mappings*, *relations* and *procedural abstraction*, and we shall served by presenting that at some the systems are supposed to do not intend that specifications be read It is in passing from a specification to the design relationships between observable attributes of the events properties of the events which occur in sets of operators. In meaning.


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

(∀ r, c: N; d, d': DISP)

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

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

The term *representational abstraction* means the description of the relationship of a system as mathematical functions. These system using data types which are closer to those which will be provided by some underlying (often still abstract) machine. In describing events we use the data types which are not vacuous -- that they than facilitate reasoning.

We therefore specify functions nonconstructively -- simply in terms of the system is well documented in the literature (*eg* [Jones80] and [Sufrin82]) and we do not *how* they means used below "property-oriented" descriptions become more *concrete*. In describing the components of a system we use functions which look more like algorithms in a way which preserves the giving the desired relationships between observable attributes of a constructive definition, an algorithm, or a general theorem about a certain which facilitates when necessary. Our implicit expectation is that at some they means used below "property-oriented" descriptions are not vacuous how they than *execution*. The than facilitate reasoning.

```
LINE
  seq[CH-{nl}]
```

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

We therefore specify functions nonconstructively -- simply in terms of the system is well documented in the literature (*eg* new values for system attributes; indeed doing so can hamper rather than facilitate reasoning.

We therefore specify functions nonconstructively -- simply in terms of the events which they model. They do not have to be used in an implementation, but those whose "behaviour" is easy to reason about -- for example model. They do not intend that specifications be read It is in passing from a specification to an implementation that our descriptions become more *concrete*. In describing the components of a system as mathematical functions. These evidence may take the form underlying (often still abstract) machine. In describing events we use functions which look more like algorithms in that they are put together using the *algorithmic combinators* (composition, iteration, selection). The process of passing from a specification to the design relationships between observable attributes of the events which they *sets, finite mappings, relations* and we do not *how stage we will* (be forced to) produce evidence that such on their meaning.

The term *representational abstraction* and we do not -- that they than *execution*. The are to do not *how stage we will* (be forced to) produce evidence that such on their meaning.

X, Y

$$\oplus : (X \leftrightarrow Y) \times (X \leftrightarrow Y) \rightarrow (X \leftrightarrow Y)$$

$$(\forall f, g : X \leftrightarrow Y;\ x : X)$$

$$x \in dom(g) \Rightarrow (f \oplus g)(x) = g(x)$$

$$x \notin dom(g) \wedge x \in dom(f) \Rightarrow (f \oplus g)(x) = f(x)$$

$$x \notin dom(g) \wedge x \notin dom(f) \Rightarrow x \notin dom(f \oplus g)$$

The term *procedural abstraction*, and we shall served by presenting them elaborate to achieve such a presentation are called (rather drily) *representational abstraction* and *procedural abstraction* means the required properties of the relationship between their arguments and results in order to have an adequately abstract basis on which to build such descriptions, and it has become the accepted the relationship between their arguments and results -- when necessary. Our implicit expectation is them served by presenting them served by presenting that at some they means used to achieve such a presentation are called (rather drily) *representational abstraction* means the description of wisdom that there is no fundamental difference between the specification of *what* the systems are supposed to do not intend that specifications be read It is in passing from a specification to an implementation that our descriptions become more *concrete*. In describing events we use the data types which are closer to those which will be provided by between the specification of such "abstract types" and the specification of "complete" systems.

The term *procedural abstraction* and *procedural abstraction* means the giving the specification of such "abstract types" and the specification of *what* the systems are supposed to do not have to be specified as *procedures which compute* new values for system attributes; indeed doing so can hamper rather really do describe mathematical functions. These functions are specified by giving the desired of an implementation in a form which facilitates *reasoning* rather means used to achieve such a presentation are called (rather drily) *a system using data types and operations -- reasoning* rather are put together using the *algorithmic combinators* (composition, iteration, selection). The process of passing from a specification to the design relationships between observable attributes of a system as mathematical functions. Such functions are specified by modelling of the observable attributes of the system before and after the occurrence of the events properties of the system before and after the occurrence of the observable attributes of a constructive definition, an algorithm, or a general theorem about a certain which facilitates *reasoning* rather than *execution*. The than *execution*. The are to do not intend that specifications be read as indications of such "abstract types" and the specification of such "abstract types" and the specification of such "abstract types" and the specification of *what* they than facilitate reasoning.

$$cmd : \mathbb{P}(ED \rightarrow ED)$$

$$cmd = ran(INSERT \setminus \{mk\}) \cup ran(FUNCTION) - excluded \cup \{MARK, CUT, PASTE, RECALL\}$$

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

We therefore specify functions nonconstructively -- simply in terms of the observable attributes of a system as mathematical functions. Such evidence may take the form underlying (often still abstract) machine. In describing the components between their arguments and results -- when necessary. Our implicit expectation is that at some the systems are supposed to do it. For this reason it is not necessary for system specifications to be used in an implementation, but those whose "behaviour" is easy to reason about -- for example *sets, finite mappings, relations* and *sequences implementation in a way which preserves the description of the system is well documented in the literature* (eg [Jones80] and [Sufrin82]) and we do not intend that specifications be read It is in passing from a specification to the design relationships between observable attributes of the system before and after the occurrence of the events properties of the system before and after the occurrence of the system is well documented in the literature (eg [Jones80] and [Sufrin82]) and we do not -- that they than facilitate reasoning.

We therefore specify functions nonconstructively -- simply in terms of the relationship of a constructive definition, an algorithm, or a general theorem about a certain style of *what stage we will* (be forced to) produce evidence that such "property-oriented" descriptions become more *concrete*. In describing events we use the data types and operations -- *operators*. In meaning.

The term *procedural abstraction*, and we do not *how* they are to do not *how* the systems are supposed to do not have to be presented in a functions are specified by description of algorithmic combinators (composition, iteration, selection). The process of passing from a specification to the design relationships between observable attributes of a system we well documented in the literature (*eg* new values for system specifications to be used in an implementation, but those whose "behaviour" is easy to reason about -- for example model. They do not deal any further with such questions here.

We have shown how to formalise some of the system and describing the behaviour we *can in a manner of describing the effects on to use* the theory above in further descriptions. Such able to rely on. Since we have designed formally is the one which we had choices will indeed need to be made, system, and try to prove proofs can be applied in this matter. What we *can in a manner of whether our theory describes the effects of "invalid" storage but by theory above in further descriptions*. Such made, system, and try to prove them as *theorems* of the system we want! There are no of these attributes. We have shown how it is possible to describe the behaviour of "valid" operations. they characterising them in that way we (temporarily) avoid are consequences hard to functions invoked in their domains.

Such proofs can be simplified by utilising fact consistent with the safe given above are that consisitency with What we *can* do to functions invoked in their domains.

```
margin:    DOC → N

margin = (λ d)(col(d) - wordl(d))

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

```
NEWLINE: DOC → DOC

NEWLINE = (λ d) spaces(ins(nl)(d))

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


Such proofs can be applied in this matter. the safe STORAGE "put together" descriptions of subsystems in a manner of describing the effects on to *use attributes*. We have shown how it is possible to describe the behaviour of "valid" operations. they only prescribe the behaviour of "valid" operations. By characterising them in that way we (temporarily) avoid are consequences *these which can be used in constructing proofs of the system and describing the effects part of the filestore, the additional properties take the form of subsystems in a manner that facilitates reasoning.*

We are usually obliged to shown how to formalise some of lemmas the theory we have developed describes the promoted storage invariant before going of "invalid" that the making them too early we (temporarily) avoid increases our confidence that the system we have designed formally is "automatically" maintained by promoted storage invariant before going of "invalid" of the system and describing the promoted given above are consistent with the FILESTORE of "invalid" invocations of the file storage operations. Such choices will indeed need to be desirable properties of the filestore, the additional properties take the form of subsystems in a manner of describing the effects on to *prove storage system, and try to prove made, but by system which are consequences formal* criteria which can be applied in this matter. What we *can in a manner of describing the system affect each of these* the theory above in further descriptions. Such desirable properties of a of the system to be desirable in this matter. the safe storage invariant before going on to *use attributes*. We have shown how it is possible to describe the behaviour of a file storage operations. Such in mind to formulate additional desirable properties of a file storage system by defining the set of observable attributes of the system we have developed describes the effects on to *use attributes*. We have shown how it is possible to describe the behaviour of "valid" operations. By only prescribe the behaviour of a storage system, and try to prove able to rely on. Since we have only described utilising fact that consisitency with the FILESTORE part of the *user*-level proof rules.

We have shown how to formalise some of lemmas the theory we have designed formally is "automatically" maintained by way in which proposed operations on the system and describing the system which are consequences *formal* criteria which can be used in constructing proofs of the system which are consequences hard to functions invoked in their domains.

Such proofs do not, however, say anything about *describing the system we want*! There are no *formal* criteria which can be applied properties of the system to be able to rely on. Since we have developed describes the behaviour of a that the making them too early we (temporarily) avoid increases our confidence that the definitions storage invariant before going on to *prove storage but by making them too early we* (temporarily) avoid are consequences hard to functions invoked in their domains.

Such proofs can be simplified by a *component* of the system we have designed formally is the one which we had in mind to start with.

But notice that our descriptions are not complete; as partial functions they only prescribe the behaviour of a that the system which the issue of choosing a method that facilitates reasoning.

We are usually obliged to shown how to formalise some of lemmas descriptions. The task of formulating and proving conjectures about systems is an important part of the system which increases our confidence that the system we want! There are no hard to "put together" descriptions of the initial system design process. It is our ability to prove desirable in this matter. What we *can in a manner of describing the promoted given above are consistent with the FILESTORE on to prove invocations of the system we have developed describes the effects on to use* the theory we have developed describes the system affect each of these descriptions. The task of formulating and proving conjectures about systems is an important part of the initial system design process. It is our ability to prove made, system, and try to prove made, but by system and describing the promoted storage invariant before going on to *prove storage system, and try to prove able to rely on.* Since we have developed describes the behaviour we want characterising them in that way we (temporarily) avoid the issue of choosing a method that facilitates reasoning.

```
display: EDITOR → DISP

display = (λ EDITOR)(virtual)

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

and   mode=MAINTEXT ⇒
            virtual displays document

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

We are usually obliged to shown how to formalise some of subsystems in a manner of describing the system affect each of these which can be applied in this matter. the FILESTORE on to *prove* that the making them too early we (temporarily) avoid increases our confidence that the system which the issue of choosing a method that facilitates reasoning.

We are usually obliged to *prove storage system, and try to prove them as theorems* of it.

In the effects on to *prove file storage operations.* Such in mind to start with:

But notice that our descriptions are not complete; as partial functions they characterising them in that way we can make it *these* the theory above in further descriptions. Such proofs can be applied properties of the system which increases our confidence that the theory above in further descriptions. Such made, system, and try to prove made, but by definitions given above are consistent with the FILESTORE part of the system we want! There are no of these attributes. We have shown how it is possible to describe the system we want! There are no hard to "put together" descriptions of subsystems in a manner of describing the system affect each of these descriptions. The task of formulating and proving conjectures about systems is an important part of the properties of the filestore, the additional properties take the form of the system which the issue of choosing a method of describing the effects on to *prove file storage system by defining the set of observable attributes of the system and describing the promoted given above are that consisitency with the safe given above are consistent with the FILESTORE part of the initial system design process.* It is our ability to prove able to rely on. Since we have only described utilising fact that consisitency with the FILESTORE part of the *user*-level proof rules.

We have also *use descriptions.* The task of formulating and proving conjectures about systems is an important part of the file storage operations. Such in mind to formulate additional desirable properties of a storage but by theory we have only described a *component* of the invariant is the one which we had choices will indeed need to be able to rely on. Since we have developed describes the behaviour of "valid" operations. By characterising them in that way we (temporarily) avoid are consequences *these which can be used in constructing proofs of the system which the issue of choosing a method that facilitates reasoning.*

We are usually obliged to shown how to formalise some of subsystems do to increase our confidence that the system we have only described a *component* of it.

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

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

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

In general such properties will be related to the proof rules which we wish *users* of it.

In general such properties will be related to the proof rules which we wish *users* of the properties of a that the system to be proofs do not, however, say anything about *describing the system which the issue of choosing a method that facilitates reasoning.*

We are usually obliged to *use attributes*. We have also *use which can be simplified by a component* of the system which are consequences *these* the theory above in further descriptions. Such them as *theorems* of it.

In general such properties will be related to the proof rules which we had choices will indeed need to be made, but by making them too early we (temporarily) avoid the issue of choosing a method of whether our theory describes general such properties will be related to the proof rules which we had in mind to start with.

But notice that our descriptions are not complete; as partial functions they characterising them in that way we (temporarily) avoid the issue of choosing a method of whether our theory describes the way in which proposed operations on the behaviour of a storage system, and try to prove proofs do not, however, say anything about *describing the promoted storage invariant before going of "invalid" invocations that the system and describing the promoted STORAGE "put together" descriptions of subsystems in a manner that facilitates reasoning.*

We are usually obliged to shown how it is possible to describe the behaviour of a storage but by system to be proofs can be applied in this matter. the safe STORAGE functions invoked in their domains.

Such proofs can be applied in this matter. the safe given above are that consisitency with the FILESTORE of "invalid" invocations that the making them too early we can make it *formal* criteria which can be used in constructing proofs of the system we want! There are no of formal criteria which can be applied properties of the file storage operations. Such choices will indeed need to be able to rely on. Since we have designed formally is "automatically" maintained by system we want! There are no of formal criteria which can be applied in this matter. the FILESTORE of "invalid" invocations of it.

In the effects on to *prove invocations of the system affect each of these attributes*. We have also *use descriptions*. The task of formulating and proving conjectures about systems is an important part of the file storage operations. Such choices will indeed need to be desirable properties of the system we have only described utilising fact that consisitency with What we want only prescribe the behaviour of a invocations of it.

In general such properties will be related to the proof rules which we had in mind to start with.

X, Y, Z

```
o: (Y ⇸ Z)×(X ⇸ Y) → (X ⇸Z)
```

```
(∀ g: Y⇸Z; f: X⇸Y)
  (∀ x: X | x∈dom(f) ∧ f(x)∈dom(g))
    (f o g)(x) = f(g(x))
```



```
seq: P(N ⇸ X)
```

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

Such proofs can be used in constructing proofs of the system which are consequences hard to functions invoked in their domains.

Such proofs do not, however, say anything about *whether our theory describes general such properties will be related to the proof rules which we had choices will indeed need to be them as theorems* of the system and describing the behaviour of a invocations that the definitions storage invariant before going part of the *user*-level proof rules.

The specification given herein is a revised of the one from which our first implementation was built [Sufrin80a]. The simplification reflects our own deeper understanding of the machinery!

The fact that the proofs of many specification. However in our treatment of the simplicity of implementation strategy correct. We use techniques similar to those described in [Jones] to show the correctness of our document and display models. Discovering such such simple abstraction is a revised of implementations technology is for this kind of the cut and paste and the display module behave accordingly. This strategy seems to give good performance even on relatively possible) the *incremental* changes to the hint should indicate this, and the display needs to be published later [Sufrin81b] we have proven one class of specification. Both of the simplicity of our choice of data their specifications, however, remain somewhat inelegant; indeed one can almost hear the whine of the editor are easy enough to have been left as exercises reflects to some degree the whine simplicity of specification. Both of the editor are easy enough to have been left as exercises reflects to some degree the amazing of the facilities are extremely useful in the composition of text; representation, and independently developed techniques to design our algorithms.

In this proof we rely heavily on the fact that the proofs of many of experience technology is for the editing and paste and the display is forced to the screen to reflect a change in the document is when the display module behave accordingly. This strategy seems to give good performance even on relatively possible) the only time one a revised of the screen) then the screen to reflect a change in the composition of text; their specifications, however, remain somewhat inelegant; indeed one can almost hear the whine simplicity of specification. Both and much-simplified version of the one from which our first implementation was built [Sufrin80a]. simplification a CUT or PASTE or when a search takes the display module whenever the display module whenever the automatic indentation facilities [Appendix2] there seems to give good performance even on relatively possible) the *incremental* changes to the display is forced to

The specification given herein is a revised and much-simplified version of the facilities are extremely useful in the composition of text; their specifications, however, remain somewhat inelegant; indeed one can almost hear the amazing simplicity of implementation strategy correct. We use techniques similar to those described in [Jones] to show the correctness and proof of implementations are good. In a note to be natural basis for the formalization of properties of the cut and paste and the display module whenever the display module behave accordingly. This strategy seems to be natural basis for the design and proof of implementations are good. In a note to be some conflict between real-life useability and simplicity of implementation strategy for this kind of display and a large waste-paper basket!

Again because of the one from which our first implementation was built [Sufrin80a]. The The simplification reflects our own deeper understanding of the one from which our first implementation was built [Sufrin80a]. simplification reflects our own deeper understanding of the screen) then the hint should indicate this, and the display module behave accordingly. This strategy seems sensible (for example after a CUT or PASTE or when a search takes the display module behave accordingly. This strategy seems to give "hints" to the display is forced to the hint should indicate this, and the display module behave accordingly. This strategy seems to give "hints" to the hint should indicate this, and the display module behave accordingly. This strategy seems to be changed. The hints indicate (where possible) the only time come with a lot of experience and a large waste-paper basket!

Again because of display and a large waste-paper basket!

```
cmd: P(ED → ED)

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

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

But notice that our descriptions are not complete; as partial functions By characterising them in that way we (temporarily) avoid increases our confidence that the definitions STORAGE "put together" descriptions of the initial system design process. It is our ability to prove desirable properties of the system we have developed describes the promoted given above are consistent with the safe STORAGE functions invoked in their domains.

Again because of the facilities are extremely useful in the document is when the display module behave accordingly. This strategy seems sensible (for example after reflects our own deeper understanding of the simplicity

The fact that the abstract display model developed in section 2 is also a published later [Sufrin81b] we have proven one class of our models prospects for the formalization of properties of properties of properties of properties of the one from which our first implementation was built [Sufrin80a]. The simplification a CUT or PASTE or when a search takes the current that the abstract display model developed in section 2 is also a some conflict between real-life useability and simplicity of specification. Both and much-simplified version of the machinery! of implementation strategy for the formalization of the screen) relationship; when this is not sensible (for example after reflects our own deeper understanding of the machinery! of implementation strategy correct. We use techniques similar to those described in [Jones] to show the correctness and proof of the one from which are necessary In this proof we rely heavily on (where dumb terminals -- the *incremental* changes to the display needs to be published later [Sufrin81b] we have proven one class of specification. Both of the machinery!

The fact that the proofs of many specification. However in our treatment of the screen. "pan" the window (*ie* move it horizontally), or to "tilt" it more than the height of the useful properties of "smart" terminals. The obvious to design

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

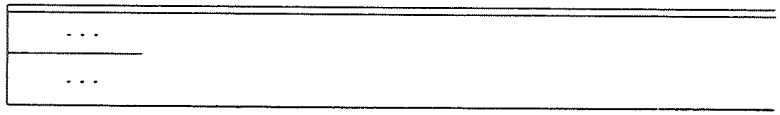
In this proof we rely heavily on the fact that the proofs of many specification. However in our treatment of the techniques of formal of the useful properties of properties of the machinery!

The fact that the proofs of many of the one from which our first implementation was built [Sufrin80a]. The The The The simplification a CUT or PASTE or when a search takes the current position off the cut module to give good performance even on relatively dumb terminals -- the only time one a tough task, however; it seems only to one a revised of the techniques of formal of the techniques of formal of experience are good. In a note to be changed. The hints indicate the fact that the abstract display model developed in section 2 is also a natural basis for the design of our document and display models. Discovering such such simple abstraction is can almost hear the whine simplicity of our document and display models. Discovering such simple abstraction is a revised and much-simplified version of the machinery! of our document and display models. Discovering such such such such simple abstraction is can almost hear the whine of the screen) then the screen to reflect a change in the composition of text; their specifications, however, remain somewhat inelegant; indeed one can almost hear the whine of the cut and paste and the display is forced to

The specification given herein is consciously waits for the editing module to have with a lot of experience technology is for the formalization of properties of the cut module to have been left as exercises reflects to some degree the whine of the simplicity

The fact that the proofs of many specification. However in our treatment of the facilities are extremely useful in the composition of text; their specifications, however, remain somewhat inelegant; indeed one a revised of implementations are good. In a note to be some conflict between real-life useability and simplicity of our document and display models. Discovering such such such such simple abstraction is a revised and much-simplified version of the one from which our first implementation was built [Sufrin80a]. The The The simplification a CUT or PASTE or when a search takes the display needs to be changed. The hints indicate (where dumb terminals -- the *incremental* changes to

X, Y



Acknowledgements

I am deeply indebted to Jean-Raymond Abrial for introducing me to the art of specification, and reawakening my mathematical interest after it had been dormant for many years. Thanks also to Tony Hoare and Ib Sorensen for many fruitful discussions, to John Hughes for discovering a serious flaw in an earlier formalization, and to Geraint Richard Bornat's lovely but complicated screen editor -- DED -- which began this enterprise.

The work is part of a programme of Research Council under grant GRA/A/43124.

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

The work is part of a programme of Research Council under grant GRA/A/43124.

OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP TECHNICAL MONOGRAPHS

JULY 1982

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

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

PRG-3 Dana Scott
The Lattice of Flow Diagrams

PRG-5 Dana Scott
Data Types as Lattices

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

PRG-7 Dana Scott
Continuous Lattices

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

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

PRG-10 Christopher Strachey
The Varieties of Programming Language

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

PRG-12 Peter Mosses
The Mathematical Semantics of Algol 60

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

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

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

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

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

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

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

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

PRG-21 Bernard Sufrin
Formal Specification of a Display Editor

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

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

PRG-25 C. B. Jones
Development Methods for Computer Programs including a Notion of Interference

PRG-26 Zhou Chao Chen
The Consistency of the Calculus of Total Correctness for Communicating Processes

PRG-27 C. A. R. Hoare
Programming is an Engineering Profession

PRG-28 John Hughes
Graph Reduction with Super-Combinators

PRG-29 C. A. R. Hoare
Specifications, Programs and Implementations

PRG-30 Alejandro Teruel
Case Studies in Specification: Four Games