25

Abstract modelling of interactive systems

D. Goldson

Department of Computer Science,
Massey University,
New Zealand.
D.Goldson@massey.ac.nz

ABSTRACT One approach to modelling interactive systems uses abstract models to define system classes. A system can then be evaluated with respect to a model that captures some generic user interface criteria. This paper describes a case study in applying this methodology to the analysis of a text editor. Criteria of predictability, observability and reachability are postulated for the editor. The paper concludes that abstract modelling is a useful design methodology that can be effectively integrated with a methodology of prototyping.

KEYWORDS Prototyping, formal methods, interaction models, software engineering.

1. INTRODUCTION

One approach to modelling interactive systems uses abstract models to define classes of systems. A particular system can then be evaluated with respect to properties of a model thought of as generic interface design principles or as normative interface criteria. From a system development perspective, this methodology offers a framework for design and decision support in which properties of an interaction model are viewed as system requirements. From a completed system perspective, it can also be used as a framework for evaluation in which model properties are viewed as evaluation criteria.

A number of abstract interaction models are proposed by (Dix, 1991), including the PIE model used in this paper. Dix identifies three kinds of property relevant to PIE systems. These are

- *predictability*:what can you predict from what you see or can potentially see?
- *observability*: what can you potentially see?
- *reachability*: what can you reach from where you are?

These suggest themselves as being, in some general sense, desirable. Predictability and observability can be interpreted as asserting a user's mastery of a system (the user always knows what will appear next, they always know what the result will be, and so on) and, in a similar way, reachability can be interpreted as expressing general usability criteria (the user will never get stuck, will always be able to get back to a certain point, and so on). The difficulty of design, of course, lies in finding the right properties for the system of interest. These principles can be used to express necessary conditions for usability that are "able to stop interfaces being fundamentally unusable, but not ensure that they *are* usable" (Dix p14). No methodology can guarantee that.

The paper describes a case study in interface analysis. The motivation for this work is pragmatic. Given the potential benefits of using abstract models, (Dix, 1991) and (Runciman, 1990), it is important to test them on realistic problems since this is the only way that their potential can be assessed. The editor, while itself simple, is a realistic analysis task rather than a merely illustrative one. The main positive outcome of the case study is the support it lends to using modelling and prototyping as an integrated methodology.

Human–Computer Interaction: INTERACT'97 S. Howard, J. Hammond & G. Lindgaard (editors)
Published by Chapman & Hall

Subsequent sections describe the case study. The focus is on HCI design principles concerned with aspects of system *behaviour* as it effects the user. The evaluation of the interface in isolation is of no interest. §2 briefly reviews Dix's PIE and red-PIE models. The editor (Goldson, 1996) is prototyped in the functional programming language Haskell (Jones, 1996) and its main features are described in §3. §4 modifies the editor to fit these models and §5 then uses them to analyse the prototype. §6, the final section, draws some conclusions.

2. *P*ROGRAM *I*NTERPRETATION *E*FFECT MODELS

A PIE model is given by an *i*nterpretation function which maps *p*rograms (command sequences) onto *e*ffects. This is about as general as a model can get: input, process, output. An instance of the model is given by specifying the type of commands over which programs are defined, the type of their effects and the interpretation function that maps programs to effects.

An alternative definition uses a `doit` function defined over system states and commands. If the effect space is identified with the state space, with a distinguished intitial state `s0`, a function `pie` from programs to effect histories is defined so that

```
pie doit s0 [c1,c2,..]     =
[s0, doit s0 c1, doit (doit s0 c1) c2,..]
```

```
pie :: (e -> c -> e) -> e -> [c] -> [e]
pie = scanl
```

The interpretation function of this PIE is just some function of its effect history.

The red-PIE model adds extra structure by distinguishing the *r*esult and *d*isplay as two components of the effect. Following (Runciman, 1990) a red-PIE can be modelled as a function `red` which takes PIE `pi`, display `dp` and result `rt` functions and returns a pair consisting of the display history and the final result of the interaction.

```
red :: ([c] -> [e]) -> ([e] -> d)
            -> ([e] -> r) -> [c] -> (d,r)
red pi dp rt = (\es -> (dp es, rt es)) . pi
```

A PIE model of a text editor defined by a `doit` function is described in (Dix p161). The effect space is the cartesian product of edited objects and displays. Commands are applied to objects or displays using a pair of state transition functions $\texttt{doit}_{\texttt{object}}$ and $\texttt{doit}_{\texttt{display}}$ and an adjustment function `adjust` is used to maintain any invariants of the object-display relationship. This is basically the same as the model used by (Sufrin, 1982) and prototyped in (Goldson, 1996), except that changes to the display are there modelled as side effects of changes to the edited object so that all commands are applied to the edited object and no commands are directly applied to the display itself. There is no role for $\texttt{doit}_{\texttt{display}}$. §4 will apply this model to the editor but to follow the details it is first necessary to describe the editor's main features.

3. A PROTOTYPE TEXT EDITOR

A full specification of the prototype is given in (Goldson). This section summarises the main features. The edited and display objects are modelled as objects of type `EDSTATE` and `DPSTATE` and an editor is got by combining these components.

```
type EDITOR = (EDSTATE,DPSTATE)
```

3.1 Documents

The edited document is a component of the inner state. It is modelled as a pair of strings. For example, the document with text `"Beautiful Soup"` and the cursor to the left of `'S'` is

```
(" lufituaeB","Soup")
```

Storing the text on the left in reverse order simplifies the definition of commands. These are modelled as state to state transformations on DOCs.

```
type DOC = ([Char],[Char])
type DCOMMAND = DOC -> DOC
```

3.2 Editor states

The full inner state supports a modal command interface in which, for some commands, the user is required to switch mode before entering the command.

```
data EDMODE     = Edit | Command | Exit
type EDSTATE    = (DOC,DOC,EDMODE)
```

In `Edit` mode edit commands effect the document (the first component). In `Command` mode they effect a command buffer (the second component). Commands are now modelled as state to state transformations on `EDSTATE`s.

```
type ECOMMAND  = EDSTATE -> EDSTATE
```

There are only two. `basic` selects the right text for editing based on the mode.

```
basic :: DCOMMAND -> ECOMMAND
```

```
basic f (d,c,m) = case m of
                    Edit    -> (f d,c,m)
                    Command -> (d,f c,m)
```

`edswitch` defines the modal commands. `switch` switches mode. `mcommand` interprets the contents of the command buffer `(ctext eds)` as an ECOMMAND and applies it to the current state.

```
edswitch, switch, mcommand :: ECOMMAND

edswitch = mcommand . switch

switch (d,c,m) =
   case m of
      Edit    -> (d,emptydoc,Command)
      Command -> (d,c,Edit)

mcommand eds =
   (case ctext eds of
     "t"       -> basic lmdoc
     "b"       -> basic rmdoc
     "q"       -> exit
     otherwise -> basic docid) eds
```

The mode also effects the way the state is displayed. In `Edit` mode only the document is displayed. In `Command` mode both document and command text are displayed. This relationship is modelled by `edisplayed`.

```
edisplayed   :: EDSTATE  -> DISP
```

To give an example, the editor state

```
((" lufituaeB",
"Soup, so rich and green,\nWaiting in a hot
tureen!"), ("",""),Edit)
```

has an unbounded display with document cursor at `(1,10)`

```
(["Beautiful Soup, so rich and green,",
  "Waiting in a hot tureen!"], (1,10))
```

3.3 Displays

`DISP` is the type of an intermediate display defined by its contents and cursor. It represents an abstract view of the edited object through an 'elastic' or unbounded display.

```
type DISP     = (CONTENTS,CURSOR)
type CURSOR   = (Int,Int)
type LINE     = [Char]
type CONTENTS = [LINE]
```

The actual display is more complicated, being of fixed size with an origin onto the unbounded display. `window` defines its contents, given its size and origin and the contents of the unbounded display.

```
type ORIGIN = (Int,Int)

type DPSIZE = (Int,Int)

window :: ORIGIN -> DPSIZE ->
                    CONTENTS -> CONTENTS
```

The display cursor is got by projecting the document cursor onto the display, counting from the origin of the display rather than the origin of the document.

```
project :: ORIGIN -> CURSOR -> CURSOR
```

The display contents, its cursor, size and origin define the state of the display.

```
type DPSTATE  = (DISP,DPSIZE,ORIGIN)
```

To give an example, a display of size `(2,40)`, origin at `(1,8)` and the unbounded display of §3.2 gives a display state with display cursor `(1,2)`

```
(((["l Soup, so rich and green,",
   "in a hot tureen!"],
  (1,2)),(2,40),(1,8))
```

The relationship between the edited object `EDSTATE` and the display `DPSTATE` is determined by an invariant `inview` that keeps the document cursor in view. `inview` is kept invariant by a scrolling function `scroll` that derives a new origin for `DPSTATE` whenever `EDSTATE` is changed.

```
inview :: ORIGIN -> DPSIZE -> CURSOR
                 -> Bool
scroll :: ORIGIN -> DPSIZE -> CURSOR
                 -> ORIGIN
```

3.4 Interaction

The editor commands are shown in Table 1. To keep the prototype simple they were restricted to insertion, navigation and deletion commands. Quiting the editor is modal, `top` and `bottom` are both modal and non-modal, and the rest are non-modal. `edswitch` switches between the two modes.

A key event is interpreted as a character insertion command unless it is escaped by `kbesc` when it is interpreted as a navigation or deletion command. The function `key` interprets the event depending on the state of the bimodal keyboard.

```
data KBMODE     = Insert | Function

key :: KBMODE -> Char -> (KBMODE,ECOMMAND)
```

Keys	*Name*	*Description*
Any printable character or `nl`	`ins`	Insert character at the current cursor position
`kbesc` `	`lmchar`	Move left one character
" 1	`lmword`	Move left to the beginning of a word
" 2	`lmline`	Move left to the beginning of a line
" 3	`lmdoc`	Move left to the beginning of the document
`kbesc` =	`rmchar`	Move right one character
" -	`rmword`	Move right to the beginning of a word
" 0	`rmline`	Move right to the beginning of a line
" 9	`rmdoc`	Move right to the end of the document
`kbesc` ~ or `delete`	`ldchar`	Delete left one character
`kbesc` !	`ldword`	Delete left to the beginning of a word
" @	`ldline`	Delete left to the beginning of a line
" #	`lddoc`	Delete left to the beginning of the document
`kbesc` +	`rdchar`	Delete right one character
" _	`rdword`	Delete right to the beginning of a word
")	`rdline`	Delete right to the beginning of a line
" (	`rddoc`	Delete right to the beginning of the document
`kbesc s`	`edswitch`	Switch editor mode
None	`lmdoc`	Move left to the `top` of the document
None	`rmdoc`	Move right to the `bottom` of the document
None	None	quit the editor
Any other key	`docid`	Do nothing

Table 1: Editor commands.

4. MODELLING THE EDITOR AS A PIE

The editor can be modelled as a simple two layer PIPE (Dix). This retains a surface view of the interaction while also taking into account that the user will interact with the editor at a deeper level than the purely surface level of the keyboard. The user *interprets* key sequences as *commands* on an *underlying* document.

The PIPE model in Figure 1 splits a system PIE `pies` into an outer PIE `piep` that maps user input at the keyboard into the commands of the inner PIE `pief`. In fact, all three are red-PIEs. The display and result of `pies` is identified with `pief` and the result of `piep` is the input to `pief`.

4.1 The functional PIE `pief`

The effect space is the editor state space. `doiteds` is the $\texttt{doit}_{\texttt{object}}$ function and defines the effect of commands on the edited object.

```
doiteds :: EDSTATE -> ECOMMAND -> EDSTATE
doiteds eds c = c eds
```

There is no $\texttt{doit}_{\texttt{display}}$ since none of the editor commands act directly on the display. There are no scroll commands for example. Of course, the display still requires adjustment to take account of changes to the edited object, so the doit function for the complete state `doitf` is defined

```
doitf :: EDITOR -> ECOMMAND -> EDITOR
doitf (eds,wns) c =
   let ceds = doiteds eds c in
      (ceds, adjust wns ceds)
```

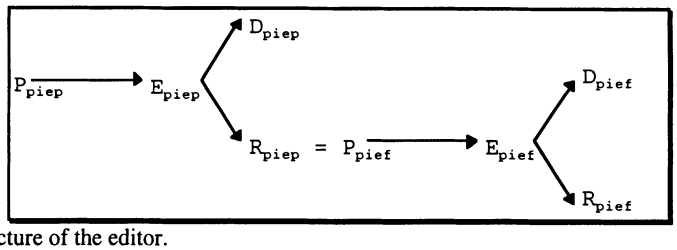

Figure 1: PIPE architecture of the editor.

```
adjust  :: DPSTATE -> EDSTATE -> DPSTATE
adjust (_,ws,o) eds =
   let (dp,dc)     = edisplayed eds
       or          = scroll o ws dc in
       ((window or ws dp,project or dc),
        ws,or)
```

The new display is got by `scrolling` to a new origin `or` on the unbounded display `dp` and then truncating this to fit the actual display at `or` and then projecting the document cursor `dc` onto this display.

The initial state is given with respect to a start up document `d`.

```
open :: RESULT -> EDITOR
ed0   = open d
```

If the quit command was non-modal the environment would signal termination to `pief` by terminating its input, and `pief` would be defined

```
pief :: [ECOMMAND] -> [EDITOR]
pief = pie doitf ed0
```

However, the quit command is modal and signals termination to the environment by changing the editor's mode to `Exit` (§3.1.2), thereby terminating output.

```
pief = takeWhile continue . pie doitf ed0

continue :: EDITOR -> Bool
continue = (/= Exit) . edmode
```

The display history and final result are projected from the effect space by a red-PIE.

```
redpief :: [ECOMMAND] -> ([DISPLAY],RESULT)
redpief = red pief displayf resultf
```

`resultf` returns the text of the edited document and forgets its cursor.

```
resultf  :: [EDITOR] -> RESULT
resultf es =
let (l,r) = eddoc (last es) in reverse l++r
```

`displayf` displays display states by inserting the cursor, filling out the contents with whitespace and framing the result.

```
displayf :: [EDITOR] -> [DISPLAY]
displayf = map (dpscreen . screen)

screen :: EDITOR -> CONTENTS
screen   = view . winst

view :: DPSTATE -> CONTENTS
view (dp,(h,w),_) =
   let ws = (h,w+1) in
   (frame ws . contents ws . dpcursor) dp
```

Here is the result of `viewing` the display state in §3.3 (`'^'` is the cursor)

```
["|------------------------------------|",
 "|1 ^Soup, so rich and green,         |",
 "|in a hot tureen!                    |",
 "|------------------------------------|"]
```

4.2 The surface PIE `piep`

`piep` models the keyboard interaction. The state space has type `(KBMODE,ECOMMAND)`, `p0` is the initial state before anything is typed and `doitp` defines the effect of a key on the current state.

```
p0 = (Insert,undef)

doitp :: (KBMODE,ECOMMAND) -> Char
                  -> (KBMODE,ECOMMAND)
doitp (m,_) c = key m c
```

The effect space is the `piep` state space and `piep` is defined

```
piep :: [Char] -> [(KBMODE,ECOMMAND)]
piep = tail . pie doitp p0
```

This is factored as a red-PIE with the mode and command histories giving the display and result.

```
redpiep  :: [Char] -> ([KBMODE],[ECOMMAND])
redpiep   = red piep displayp resultp

displayp :: [(KBMODE,ECOMMAND)] -> [KBMODE]
displayp  = fst . unzip

resultp  :: [(KBMODE,ECOMMAND)]->[ECOMMAND]
resultp   = snd . unzip
```

4.3 The system PIE `pies`

`pies` joins the two red-PIEs by making the result of `redpiep` the input of `redpief`.

```
pies :: [Char] -> ([DISPLAY],RESULT)
pies = redpief . resultp . redpiep
```

5. ANALYSIS OF THE EDITOR

This section shows how the editor's interface can be explored by postulating reachability, predictability and observability properties for it. Since the model is two level it is reasonable to examine each level for these characteristics. Starting with the inner level, let `if`, `rif` and `dif` define the effect, result and display of a `pief` interaction

```
if  = last . pief
dif = last . fst . redpief
rif = snd  . redpief
```

Every program `p` implicitly defines an effect `if p`, result `rif p` and display `dif p`.

5.1 Reachability

`pief` is reachable if and only if you can get from any state `p` to any state `q` using a strategy `s`.

```
∀p,q ∃s if (p++s) = if q                    (R)
```

`pief` is strong reachable if and only if reachable states are indistinguishable.

```
∀p,q,s,r (if (p++s) = if q ⇒
          if (p++s++r) = if (q++r))    (SR)
```

`pief` is both reachable and strong reachable. This is not surprising since the effect space is the entire system state. ((SR) is a consequence of (R) and (EP) below.) The concept of reachability is related to the concept of functional completeness of commands but it does *not* define it. Think of deleting to the end of a DOC

```
rddoc ("","Beautiful Soup") = ("","")
```

and then restoring the state with

```
[ins 'B',…,ins 'p']
```

This is no reason to leave undo out of the command set.

5.2 Predictability and observability

The easiest property to state is effect predictability or monotonicity. Two equal effects `p` and `q` are effect predictable if and only if future effects are unaffected by how `p` and `q` are reached.

```
∀p,q,r (if p = if q ⇒
                if (p++r) = if (q++r)) (EP)
```

`pief` *is* effect predictable but this is not interesting because the effect type is artificial. The user can observe the display and potentially observe the result but can not directly observe other important components of the effect, such as the display origin and the mode.

Only the display is directly observable so what can be predicted from this? Display predictability, along the lines of (EP), is out of the question, making equal displays imply equal documents. In fact, even mode predictability is too strong. Two displays that look the same may not have the same mode.

```
∃p,q (dif p = dif q &
      edmode (if p) ≠ edmode (if q))
```

Given the current display, we don't necessarily expect to know where we are, or what we've got, or even what mode we are in, but we do expect to be able to find these things out. Observability principles are needed to connect a display with other 'possible' displays, as well as other aspects of the effect, such as the mode and the current result. Since none of these are directly observable, strategies, such as document browsing, are needed.

An obvious strategy to reveal the mode from the behaviour of the display is to 'switch mode and see what happens'. Two displays that look the same after switching mode have the same mode.

```
∀p,q (dif (p++[edswitch]) =
      dif (q++[edswitch]) ⇒
      edmode (if p) = edmode (if q))  (MO)
```

For observing the result we are interested in all possible displays of `p`, defined by a characteristic function `(dreach p)`.

```
∀p::[ECOMMAND],∀d::DISPLAY
  (dreach p d ⇔
  ∃s (dif (p++s) = d & rpassive s)) (DR_df)
```

`d` is a possible display for `p` if and only if `d` is the display of a state that is reachable from `p` using some strategy `s`. The change to `p` must be passive and only alter 'allowable' aspects of its state. This gives a useful notion of display reachability. Two states `p` and `q` are display reachable if and only if `q`'s display is a possible display for `p`, `dreach p (dif q)`.

What is allowable may vary but the change must at least preserve the result and so `s` must at least be result

passive. Changing p's state to match q's display does not alter p's result.

```
∀p,s (rpassive s ⇔
      rif p = rif (p++s))                    (RP_df)
```

Result connectedness states that if two states yield the same result, whatever *is* observed of the one can be observed of the other. They should be display reachable.

```
∀p,q(rif p = rif q ⇒
                    dreach p (dif q))   (RC)
```

Of course, the converse is not true. Displays that can be made equal, by some strategy, do not imply equal results. This requires a stronger notion of display equivalence. Two states p and q are display equivalent if and only if they have the same displays.

```
∀p,q::[ECOMMAND],∀d::DISPLAY
  (dequiv p q ⇔
   dreach p d ⇔dreach q d)                  (DE_df)
```

Result observability states that observationally indistinguishable or display equivalent states yield the same result.

```
∀p,q (dequiv p q ⇒ rif p = rif q)     (RO)
```

In fact the equivalence follows from (RC) since if two states yield the same result, whatever *can* be observed of the one can be observed of the other.

```
∀p,q (rif p = rif q ⇒ dequiv p q)
```

Predicting future results based on observations requires a stronger notion of equivalence than dequiv. Intuitively, future results can only be predicted if both the mode and the *document* cursor can be worked out by observation. Any strategy to move from p to q must align document cursors with a resultant change to the display *only*. This is captured by a state equivalence stequiv that requires equality of the 'inner' state.

```
∀p,q(mequiv p q ⇔
     edmode (if p) = edmode (if q)) (ME_df)

∀p,q(docequiv p q ⇔
     eddoc (if p) = eddoc (if q))    (DE_df)

∀p,q(stequiv  p q ⇔
     mequiv p q & docequiv p q)      (SE_df)
```

We are interested in the reachability of q from p, via s, that makes the states internally the same without disturbing p's result.

```
∀p,q,s (streach p q s ⇔
      rpassive s & stequiv (p++s) q) (PR_df)
```

Two state equivalent effects are result predictable if and only if future results are unaffected by how they are reached.

```
∀p,q,r,s (streach p q s ⇒
          rif (p++s++r) = rif (q++r)) (RP)
```

In other words, you can predict the effect on the result of a program on a state p if you have a strategy to work out p's mode and where you are in the document.

5.3 piep and pies

Reachability at the piep level is about the users access to inner level commands. Any pief program qf should be available from any state pf of the pief you have reached. If you have got to pf using piep commands pp then there should exist piep commands qp that give you qf.

```
∀pf,qf::[ECOMMAND],∀pp::[Char],∃qp::[Char]
  (piep pp = pf ⇒ piep (pp++qp) = pf++qf)
```

At the pies level the keyboard mode is not directly observable from the pies display. However, it can be observed using a strategy analogous to (MO): 'type a command key and see what happens'.

```
∀p,q (dis (p++['s']) = dis (q++['s']) ⇒
      dip p = dip q)                    (KO)
```

where dis and dip are the displays of pies and piep

```
dis = last . fst . pies
dip = last . fst . redpiep
```

The effect on the display of typing `'s'`

```
+------------------------------------+
|Beautiful ^Soup, so rich and green, |
|Waiting in a hot tureen!            |
|                                    |
|                                    |
+------------------------------------+
```

depends on the mode of the keyboard. Either the mode of the editor is switched or the character is inserted.

```
+------------------------------------+
|Beautiful                           |
|>                                   |
|^                                   |
|<                                   |
+------------------------------------+
+------------------------------------+
|Beautiful s^Soup, so rich and green,|
|Waiting in a hot tureen!            |
|                                    |
|                                    |
+------------------------------------+
```

6. CONCLUSION

This section briefly assesses the PIE approach under the four headings of architecture, dynamics, prototyping and development.

The PIPE model makes the limitations of the prototype's architecture clear. Since all user interaction is directed at the edited object, and no component of the display is used to define it, the architecture is not suited to indicative input where the display is central to the interpretation of input. In the PIPE model commands are determined by their content rather than their position.

The ability to model system *behaviour* carries more positive benefits. This is *the* essential component of interface analysis and all three notions of reachability, predictability and observability are strongly behaviour oriented. Dix argues that system dynamics "are difficult to describe [but] often crucial to the feel and effectiveness of the system" (p271). Furthermore, if "the relation between the functional core `[pief]` and the system as a whole `[pies]` is *not* as a *component*, but as an *abstraction*" (p171) then the evaluation of an interface in *isolation* is a meaningless notion, since there is no interface to evaluate.

The case study supports a methodology that combines modelling and prototyping. Prototyping has a useful role in the validation process in bridging the formality gap and providing the user with an opportunity for involvement. It can also play a useful role in clarifying ideas during design (Henderson, 1986). The functional programming paradigm is well suited to the demand for rapid prototyping in the development cycle, (Henderson, 1986), (Hughes, 1989), (Runciman, 1990), and while the integration of prototyping and modelling is not without problems, creating a tension between the prototype as both specification and evaluation tool for example, the case study shows that their combination can be effective.

Finally, the use of abstract modelling gives the designer a useful framework for analysis. The methodology focuses on key abstractions, such as the effect, result and display spaces, and identifies relations between them that are important for the user. Although the red-PIE model is architecturally too general for direct manipulation systems, it can still give useful insights into behavioural properties that are relevant to all interactive systems.

The process of applying a model is likely to foster a deeper understanding of a system as the designer is forced to consider the implications of their decisions for the user. In analysing for reachability the designer is forced to consider functional completeness and the relations between different system layers. With predictability and observability, the focus is on aspects of state that effect the interaction (cursor position, mode and command buffer, window origin) and whether these can be observed.

Observability principles were given in terms of strategies, what the user would have to do to find out x, but, in all but the simplest cases, such as `(MO)` and `(KO)`, where a strategy is obvious, the strategies were left implicit. The principles say what would have to be true of a strategy for the principle to hold. For example, in `(RC)`, it must move the user from one display to another. As more is demanded of the observability of the system, in `(RO)` and `(RP)`, the requirements on these strategies become more stringent. The usefulness of the principles lie in the insights into the system that are gained in framing them but there is no expectation that the designer should *prove* them true.

Modelling the editor was done as an exercise in how modelling and prototyping can be integrated. The editor itself is perhaps too simple for the analysis to turn up any real insights, though the PIPE model did highlight a useful modularisation of the original program to give the two layer `piep` and `pief`. The model was found to be usable, useful and compatible with a prototyping methodology. Future work will look at modelling direct manipulation systems in the same way.

7. REFERENCES

Dix, A. (1991) Formal Methods for Interactive Systems, Academic Press.

Goldson, D. (1996) Functional Prototyping of Interactive Systems, Technical Report, Department of Computer Science, Massey University, New Zealand.

Henderson, P. (1986) Functional programming, formal specification, and rapid prototyping, in *IEEE Transactions on Software Engineering*, **12**, 2.

Hughes, J. (1989) Why Functional Programming Matters, in *The Computer Journal*, **32**, 2.

Jones, M. (1996) The Haskell User's Gofer System, available from `http://www.cs.nott.ac.uk/Department/Staff/mpj/hugs.html`.

Runciman, C. (1990) From Abstract Models to Functional Prototypes, in *Formal Methods in Human-Computer Interaction* (ed. M. Harrison, H. Thimbleby), CUP.

Sufrin, B. (1982) Formal Specification of a Display-Oriented Text Editor, in *Science of Computer Programming*, **1**.