Formal Methods for Interactive Systems © Alan Dix 1991
http://www.hiraeth.com/books/formal/

CHAPTER 7

Opening up the box

7.1 Introduction

When we have considered PIEs and the related models, we have been eager to give behavioural definitions of all properties of interest. We have deliberately made no assumptions about the way a system is constructed, instead concentrating fully on what can be observed from the outside. This is the essence of the black-box view of the interface. A system is described purely in terms of the keystrokes that go into it and the pixels on the screen and ink on the paper that come out. It is a safe route to take, yet it is somewhat extreme, and perhaps over-zealous. Users will not only infer information from the system (captured already in the idea of observable effect), but will also have some idea of the structure of the system. In particular, they may deal with the system at a conceptual level deeper than that of keystrokes. For example, when using many operating systems, the user's notion of command is the entire line, rather than keystrokes. We are reminded of the various interface layers suggested by other authors. Foley and van Dam (1982) refer to three major layers: lexical, syntactic and semantic. The user will be working at or close to the semantic layer: hence it is reasonable to model the deeper layers of the system. This is equally valid for output as well as input, and we should consider the user as using commands on the underlying objects of interest rather than keystrokes on the display. Referring back to the black box, we are saying that it is acceptable, when we are moving towards a valid user model, to "open up" the black box.

Another reason for opening up the black box is in order to refine the model in order to work towards an implementable specification. In many ways the two aims follow each other, as it seems a good idea to follow the intended user model in the detailed specification, even if later on this needs to be transformed somewhat to provide an efficient implementation (see Chapter 11).

As we've said, the system will be layered with the "actual objects" being manipulated at the bottom, accessed via one or more levels of interface. Examples of such layering are:

Input

- Menu selection. Although this involves several user actions (choose menu, open menu, make selection), it is perceived as one "real" action.
- Control-key sequences. For instance, in Wordstar "^KB" is not perceived as two actions but as one, "mark block beginning".
- Command interfaces. As we've mentioned already, entering a command may involve a complex sequence of typing and line editing operations; for a while, the command line itself appears to be the object of interest, but at a different level entering a command is treated as a single atomic action.

Presentation

- Framing. When framing a portion of a text to fit on a display screen, the user is aware that the screen is not the "real" object, but just a view into it.
- Pretty printing and structure editors. At least the idea is that users should see beyond the textual form to the underlying structured object.
- Folding mechanisms. Again the user should see through these to the "real" object.

As users' experience with a system grows, they may understand more of its structure. Let's imagine a simple word processor. Very naive users may even be worried that text scrolled off top or bottom of the screen is lost. Most users will be happy with the idea of a screen window, but will still have a rudimentary idea of what sort of editing and formatting is possible. A little more experience and they will realise that the form that appears on the screen is not necessarily identical to what is printed, and they will become happy with the use of different fonts and complex embedded codes for detailed control of the printed form. Still later they may be aware that both interactive and printed forms are derived from a stream of characters, some printed, some invisible; they will use newlines just like any other character in search/replace operations, perhaps know that centring is caused by an invisible control sequence, and be able to uncentre text by deleting these.

This amount of layering may be thought of as poor design, and many modern systems deliberately try to avoid this sort of scenario, making the system model as close to the interface model as possible. (Hutchins *et al*. 1986) Of course, we can never entirely get rid of layering – at least the finite interface will form a layer – and where systems retain generality and power there is almost always a more complex expert's model. The main improvement is not so much the removal of such layers. It is clear that many systems fail in their layering

because the underlying model is implementation driven and the surrounding interface serves unsuccessfully to hide this. A careful, consistent and methodological design of such interface layers based on an intended user model is what is really required.

The rather abstract discussion of relations between PIEs at the end of Chapter 2 forms a basis for considering such layered systems. In the present chapter we take instead the red-PIE as our starting point. We take the result map as being synonymous with the objects of interest, and consider two layered models where the display is a facet of the outer presentation layer, and the result of the inner layer. We use the various abstract relations to model this, but eventually become stuck. We obtain adequate declarative requirements for the relation between the object layer and the display layer, but we find it hard to describe what is in the display layer. We also fail to give more constructive methods that will be of use when moving towards a particular interface design.

This is followed by a section elaborating the model to include separate but linked states for the underlying object manipulation and the display mapping: this uses a *doit* function (state transition) at the two levels, with the outer (display) *doit* defined in terms of the inner (object) *doit* and additional functions. This therefore yields a method of constructing interfaces for existing conceptual-layer object editors.

We then consider the drawbacks and limitations of the line of modelling taken here. In particular, we see that we need models that encompass *display-mediated* interaction. This serves as an introduction to Chapters 8 and 9.

Finally, we look at the use of *oracles* as a way of describing more complicated systems whilst still retaining a linear interface architecture.

7.2 Modelling editors using PIEs

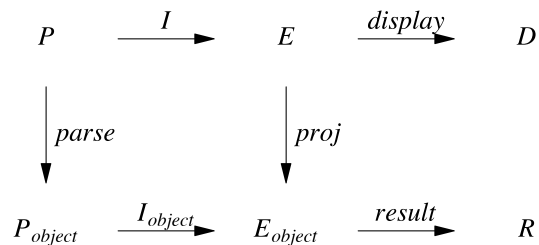
7.2.1 Basic relation between layers

We now attempt to produce simple two-layer models of interactive systems. The intention is to open up the red-PIE model. The inner level describes the functionality of the objects of interest, taken to be the result. It consists of a monotone PIE $< P_{object}, I_{object}, E_{object} >$ together with a result map $result : E_{object} \to R$. Remember that the display will only be a facet of the outer layer. The outer layer consists of the red-PIE as available to the user, $< P, I, E >$, with display map $display : E \to D$ and result map $result' : E \to R$. For any sequence of commands at the outer level, we can identify the sequence of virtual commands to which these correspond at the inner level. This can be represented by a map *parse*. If we want to retain the functionality of the inner PIE, we need to have access to all possible command sequences in P_{object} and

hence *parse* must be surjective. Similarly, we can abstract from the total state (E) the state pertaining to the underlying object (E_{object}) using an abstraction mapping *proj*. The result will be derivable completely from E_{object}. This, of course, leads to an *abstraction* relation, as defined in the previous section, between the two PIEs. If the construction is to make any sense at all, the result of the whole system must be the result obtained from the underlying objects. That is:

$$result' \quad = \quad result \circ proj$$

We can think of this either as a requirement of the system, or more constructively as a definition of *result'*. Either way, from now on we will ignore the result map from E and assume the result is obtained using the underlying result function. We thus have the following picture:



An example of this is the Unix calculator "bc". It is implemented using a simpler stack-based calculator "dc". It works by issuing commands to dc, and for any sequence of commands issued to bc there is a sequence that has been parsed and sent to dc. We can see dc therefore as an abstraction of bc. However, this is not a very good example as few users of bc would see themselves as using dc, and it is more a feature of the implementation than of the interface design. We shall see better examples later. The above diagram is very important: all the rest of the models in this chapter and much of the succeeding two chapters will be just different ways of expanding on it.

Why does it need further expansion? It already describes adequately the relation between the inner and outer functionality. We can therefore demand that our various statements hold at whichever level is appropriate. For instance, we would expect the inner PIE to be reachable, but we may not be so worried about the outer one. Note that unless the *parse* function is trivial E_{object} is different from, but related to, $R^{\dagger}$, and thus this is different from the statements we made about red-PIEs. In particular, in a mouse-based system, the commands at the inner level generated by mouse clicks will be dependent on the present display frame: thus $R^{\dagger}$ would contain this information, but E_{object} need not. Further, we

see that it is more likely that we will be able to have strategies strong passive with respect to E_{object} than $R^{\dagger}$, and this then becomes a possible desirable feature of an editor.

What this picture does not tell us much about, though, is the *extra* state needed at the outer level. This is bundled up with the object state in E and we have no separate access to it as we do for E_{object}. Any abstraction sufficient to allow the factoring of the display map will (if the system is at all observable via the display) contain all, or nearly all, of the complete system state. If we are interested only in specifying that a system satisfies certain basic functionality this is fine. If, on the other hand, we are interested in principles relating to how the display map behaves, or alternatively in building the outer PIE, perhaps in a generic way, from the inner one, we will need a model telling us more about the display.

Before we go on to consider these more elaborate constructions, we will consider some of the problems of interpretation that arise when we start to deal with principles stated at multiple levels.

7.2.2 Choice of level, the designer's freedom

While we were taking the commands to be at the outermost, physical level there was not much problem of interpretation. Given any system we could say whether it satisfied a given principle in an automatic fashion. If the principles were to form part of a usability contract then the client could be sure of the product, and the designer would have little leeway in interpreting the contract. As soon as we move on to principles at deeper levels of abstraction, life becomes far more complex.

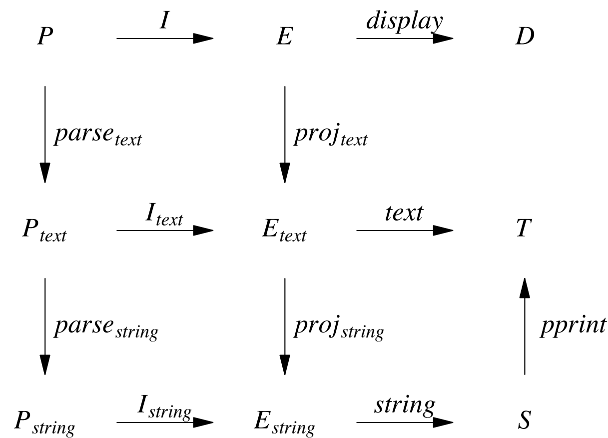
It may be that we start out with the inner PIE and the designer's job is to envelope it in an interface. It is easy then to draw up the above diagram. However, even then the designer may complain that the inner PIE has too much "interface" in it already. This is connected to the general problem of whether the interface can be or should be "separable" from the underlying application, a premise central to the development of UIMS and discussed for instance by Cockton (1986).

If there is no such fixed starting point, then life becomes far more complex. For instance, there are glaring loopholes that the designer could unwittingly fall into, or deliberately exploit. For instance, the *parse* and *proj* functions could both be trivial, leading to the original red-PIE. Alternatively, the *parse* function may be badly chosen, not picking out a sensible syntax. For instance, in a command-based system the sensible *parse* function would pick out complete lines punctuated with "newline" characters, and then pass these on. However, we could just as easily have defined a *parse* function that punctuated using the character "x". The underlying applications interpretation function I_{object} would

have to be fundamentally different. Happily, these scenarios lead to far more complex inner applications and are thus likely to be spotted. Similar, but more subtle, problems are not so easy to deal with. For instance, if we are designing a text editor, should the inner application know about a cursor and have cursor movement commands, or should it be based around context-free commands such as "insert 'a' at the 3rd column of the 10th line". Again, if we are dealing with natural language, should we be parsing at the level of words or sentences?

We have to trust to a large extent the designer's discretion in choosing the appropriate abstractions, and applying the appropriate principles to them. By asking the designer to specify which abstractions are regarded as important and what principles are to be applied, we cannot force a good design, but we are thus approaching a formal methodology which can guide the production of quality systems.

We also should expect the system to have several different layers of abstraction, recalling again the multiple layers advocated in the interface design literature. So for instance, the editor described in Chapter 11 has three such layers of abstraction: the top, full functionality layer; the text layer, where we consider unbounded formatted texts and which also includes the cursor; and a deepest string layer, where the objects are unbounded, unstructured strings of characters, and the commands are context-free (that is, no cursor). The "result" of this deepest level (obtained with a map *string*) is related to the result of the text level (the map *text*) by a pretty print function, a relation which is preserved by the abstraction:



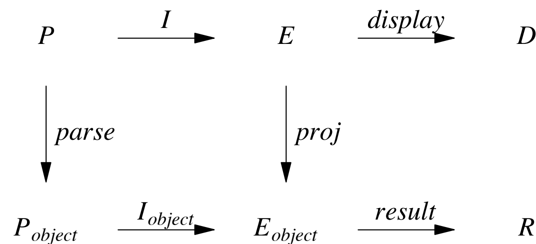
The condition on *string* and *text* is:

$$pprint \circ string \circ proj_{string} = text$$

The notion of pointer spaces introduced in the next chapter will make it particularly easy to design states satisfying this condition. However, that is jumping ahead a little.

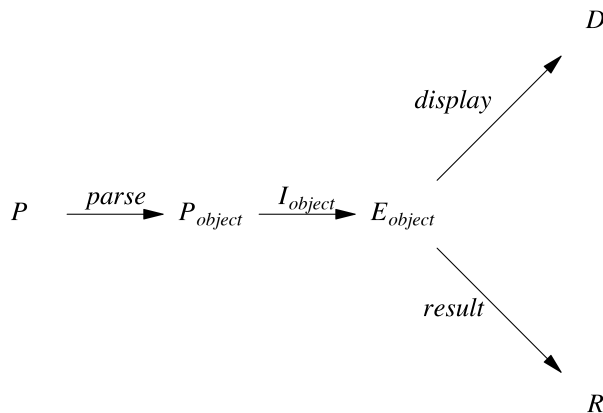
7.2.3 PIPEs

We now return to the basic two-layer abstraction in the hope of describing something more of the upper layer:



One reason for looking first at relations between PIEs at a formal level, rather than introducing them as needed, was to build discrimination between the various, quite similar diagrams and constructions we will encounter. In particular, almost invariably, when looking at the above diagram for the first time, it is read as a construction for $< P, I, E >$, or in terms of PIE relations, as an *implementation*. That is, the direction of the *proj* map is effectively reversed. The distinction is very important: the abstraction relation given above is very general, and can be applied to almost any system, whereas the implementation relation is far less general. However, as it certainly appears to be useful, we will see how far we can get with it.

Reversing the projection effectively means that the information in E_{object} is sufficient to define the display. Thus rather than proliferating function names, we can assume that the display can be obtained directly from E_{object} via the map *display*. The input layering (*parse*) and the output layering (*display*) are thus distinct, and the whole structure is a lot more linear:

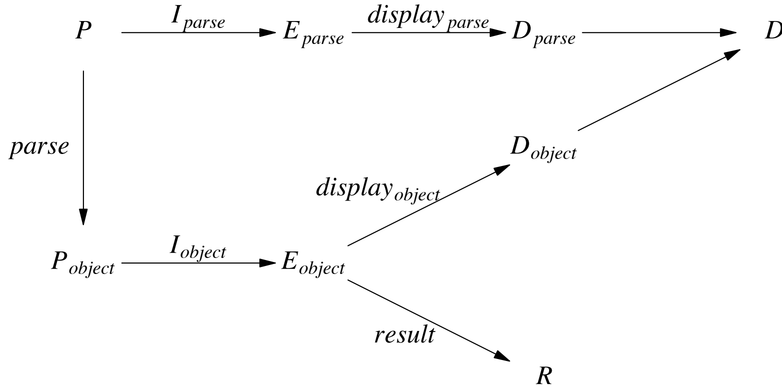


Certain kinds of input layering would just about be described by this structure, for instance, if the relation between keystrokes and object commands were one of simple macro expansion, each keystroke giving rise to one or more commands. It also describes some slightly more complex forms of input layering. For instance, in the editor "vi", keys can be prefixed by a number. This is interpreted as repeating the command that many times. Thus "5j" gets translated into the sequence of commands "DOWN DOWN DOWN DOWN DOWN". In fact, we could have a lot more complex parsing if we wished. However, as in the case of vi, this is at the expense of predictability. In vi, when we have typed the "5" there is no visual indication at all. The only way to discover that the "5" has been entered is to type something else and see what happens. The system is unpredictable. If we choose to implement any system as a PIPE with a parse function of any complexity this is bound to be the result. Unless the output of *parse* contains sufficient information to generate a display for it, there is no way this information can become visible to the user.

Clearly the parse function requires a "short cut" to the display if it is to be both powerful and predictable. We shall try to extend our model to cover this now.

7.2.4 Observable parsing

We need some sort of additional display information from the parsing to be combined with the display from the object in order to give the entire display. If we call the entire display domain D and the individual displays D_{parse} and D_{object}, then D_{object} will be obtained by a simple PIPE, and D_{parse} will have to be obtained using a separate PIE $< P, I_{parse}, E_{parse} >$ with an associated display function, $display_{parse}$:



For this to be predictable, we need to be able to observe the state associated with the *parse* function through $display_{parse}$. We can define this state by quotient on P using an equivalence relation:

$$
\begin{aligned}
p \equiv_{parse} p' \quad &\hat{=} \\
\forall q \in P \quad &\exists\, q' \in P' \\
&\textbf{st} \quad parse(\ p; q\ ) = parse(\ p\ ); q' \\
&\textbf{and} \quad parse(\ p'; q\ ) = parse(\ p'\ ); q'
\end{aligned}
$$

So two command histories are equivalent by $\equiv_{parse}$ if equal extensions to them yield equal extensions to their parsed value. We can then phrase predictability in terms of the generated PIE with interpretation $I_{\equiv_{parse}}$. The parsing is predictable if the red-PIE, with display interpretation $display_{parse} \circ I_{parse}$ and result interpretation $I_{\equiv_{parse}}$, is result observable (§3.3). Since this interpretation is monotone the red-PIE is also result predictable.

We also want to preserve the reachability of the inner PIE. This is clearly achieved if given any command history p and its parse p', then given any desired extension to p', we can find an extension to p yielding it. That is:

$$
\begin{aligned}
&\forall p \in P,\ p' = parse(\ p\ ),\ r' \in P_{object} \\
&\quad \exists\, r \in P \quad \textbf{st} \quad parse(\ p; q\ ) = p'\ ; r'
\end{aligned}
$$

or, in terms of the above red-PIE, $I_{\equiv_{parse}}$ is strong reachable. This reachability condition applies equally to the simple PIPE.

Consider the example of a command line parser. We already have some application defined by PIE_{object}. Its command set C_{object} consists of complete command lines (strings of characters with no newlines in it). We want to define a parse function, and an associated PIE_{parse} to interface this PIE, yet retain predictability. We define *parse*, I_{parse} and $display_{parse}$ thus:

$$
\begin{array}{lll l}
parse(\ null\ ) & = & null & \\
parse(\ p::c\ ) & = & parse(\ p\ ) & c \neq newline \\
parse(\ p::newline\ ) & = & parse(\ p\ ) :: I_{parse}(\ p\ ) & \\
 & & & \\
I_{parse}(\ null\ ) & = & null & \\
I_{parse}(\ p::c\ ) & = & I_{parse}(\ p\ ) :: c & c \neq newline \\
I_{parse}(\ p::newline\ ) & = & null & \\
 & & & \\
display_{parse} & = & identity &
\end{array}
$$

The full display could then be made by adding the result of $display_{parse}$ to the bottom of the result of $display_{object}$. It is fairly clear that such a construction preserves any predictability and observability properties of the original system. Of course, it is a bit simplified, having ignored command line editing, finite line buffers, etc., but such a full example would follow along exactly the same lines and conserve properties similarly.

In this example the state and display for the parser are synonymous; in other words, the PIE between P and D_{parse} is monotone. Although we'd clearly like it to be always tameable, the monotonicity is not necessary. For instance, the parser could handle a keystroke macro facility, the macro definitions being browsed and edited using the parser PIE. Going further, the typical LOGO interface could be thought of as just such a facility built over a basic functionality of turtle graphics. That is, the commands in C_{object} would be the turtle graphics commands (FORWARD, TURN, PEN DOWN, etc.) and D_{object} would be the actual graphics produced together with the turtle. The result could be the same as D_{object} or perhaps just the graphics without the turtle. All of the LOGO programming language, including all the editing of LOGO procedures, would be part of the *parse* function. The example is a little extreme, but is certainly possible.

In addition to using the "short cut" of I_{parse} for making *parse* predictable we might hope that it could exercise control over the display of D_{object}, for instance screen scrolling. This is certainly the case with the red-PIE window manager, which can be thought of as supplying a parse function with associated display for each of its windows. The combination function is rather more complex than in the previous examples, as it sometimes hides parts of the object's display.

In general, though, we can supply little in the way of display control through this "short cut". This is because display control demands a lot of knowledge about the underlying application, and any display control through the short cut would need to mimic a large part of the functionality of the application. For instance, if D_{object} was taken to be an entire formatted version of the object of interest, we might hope that information in D_{parse} could control the part of D_{object} displayed in D. However, I_{parse} would not know where the cursor was

and I_{object} would not know where the display frame was, and thus the system could not maintain the visibility conditions for local commands described in §3.4.

It would clearly be useful to describe the display and object components in a way that preserves some sort of *independence* between them. In the next section we consider a scheme that allows just such independence.

7.3 Separating the display component

In this section we consider a simple model where the application and display parts of the state are separated. This has been given a section of its own because the model is fundamentally different from those presented before. In all of these we could represent the relationships in terms of function diagrams. In the following we will have to move to a state-transition-oriented approach to capture the effect of the object on the possible display states. This is a more detailed approach to the semiformal model used to describe non-deterministic intermittent update in the previous chapter.

Consider a simple text editor. The display state may be affected both directly, by user-level commands directed at it (e.g. scroll up window), or indirectly because of changes in the object state, (e.g. changing the display frame as the cursor moves out of view). Further, even the direct commands may have their effects modified by the object state (does the scroll hide the cursor?)

7.3.1 Basic model

We capture this behaviour by having two parts to the effect, E_{object} and $E_{display}$, the total effect being the product of these two. (We assume that any parsing has taken place at a different layer.) These have associated state-transition functions, except that the display's state-transition function would depend on the object state as well as the old display state (to account for the limitations on direct display commands):

$$
\begin{aligned}
doit_{object}: &\quad C \times E_{object} \rightarrow E_{object} \\
doit_{display}: &\quad C \times E_{display} \times E_{object} \rightarrow E_{display}
\end{aligned}
$$

Ideally we would like these to be totally independent, but the display updates will need some information about the object. Later we will see how this can be minimised.

Further, we require an adjustment function *adjust* that restores any invariants, such as always displaying the current cursor line, broken after object commands:

$$adjust: \; E_{display} \times E_{object} \rightarrow E_{display}$$

Finally, we have the usual *display* and *result* mappings:

$$display: \quad E_{object} \times E_{display} \rightarrow D$$
$$result: \quad E_{object} \rightarrow R$$

Doit is defined by applying the two state-transition functions, then restoring the invariants using the adjustment function:

$$doit: \; C \times E_{object} \times E_{display} \rightarrow E_{object} \times E_{display}$$

$$doit(c, e_o, e_d) \;=\; e_o', e_d'$$

where

$$e_o' \;=\; doit_{object}(c, e_o)$$
$$e_d' \;=\; adjust(doit_{display}(c, e_d, e_o), e_o')$$

Typically we would expect commands to affect either the display only or the object only, that is:

$$\forall \; c \in C$$
$$\textbf{either} \quad \forall \; e_o \in E_{object}: \quad doit_{object}(c, e_o) = e_o$$
$$\textbf{or} \quad \forall \; e_d \in E_{display}: \quad doit_{display}(c, e_d) = e_d$$

However, in the former case the display state may (and often will) be changed *indirectly* by the adjustment function.

7.3.2 Static and dynamic invariants

Sufrin's Z specification of an editor (Sufrin 1982) fits this model except there are no direct display commands: the display is always changed by side effect and this means there is no $doit_{display}$ function. Further, it does not define the adjustment function explicitly, just the invariants it must supply. It specifies a *static invariance* predicate:

$$invariant_{static}: \quad E_{display} \times E_{object} \rightarrow Bool$$

and demands that any editor that follows the specification must have an adjustment function that satisfies:

$$adjust(e_d, e_o) = e_d' \quad \Rightarrow \quad invariant_{static}(e_d', e_o)$$

In Chapter 6 we discussed the concept of static and dynamic display invariants in fairly loose terms in order to discuss non-deterministic intermittent update. The above is, of course, part of the formal definition of a static display invariant over this sort of display model. We might want to demand additionally that the display state-transition function (if one is needed) also satisfies an appropriate condition:

$$invariant_{static}(e_d, e_o) \quad \textbf{and} \quad doit_{display}(c, e_d) = e_d' \\ \Rightarrow \quad invariant_{static}(e_d', e_o)$$

That is, $doit_{display}$ preserves the invariant.

Alternatively, we could leave *adjust* to handle this. However, if we regarded a direct display command that would violate a static invariant as an exception, we would want it to satisfy an exception principle like "no guesses"; this would be hard to ensure, so it would seem sensible to embed the knowledge in $doit_{display}$ also. Further, we could decide not to apply the adjustment function in the case of direct display commands (assuming they can be identified). Clearly, there is a bit of a trade-off here as the static invariant is spread around.

The Sufrin editor also defines a condition equivalent to display inertia. This is a *dynamic invariant*. We can similarly define a dynamic invariant as a predicate:

$$invariant_{dynamic} : \quad E_{display} \times E_{object} \times E_{display} \rightarrow bool$$

such that the adjustment function must obey:

$$adjust(e_d, e_o) = e_d' \quad \Rightarrow \quad invariant_{dynamic}(e_d, e_o, e_d')$$

The invariants would tend to precede the definition of the display state-transition function and the adjustment function in the design of an interface.

7.3.3 An example

We will now consider a simple example. We assume that $doit_{object}$ has already been defined, and further that E_{object} consists of two parts, a text consisting of a sequence of fixed-length lines, and a cursor position as line and column number, which is always within the boundaries of the text. We don't care what the object commands are or how $doit_{object}$ behaves. We define the display state to consist of a line number at which the display will start. That is:

$$E_{object} = Line^* \times \mathbb{N} \times \mathbb{N}$$
$$E_{display} = \mathbb{N}$$

In the object state, the first number is taken to be the line number of the cursor and the second the column.

The display will consist of a fixed number (25 say!) of lines and a cursor position similar to the text:

$$D \;=\; Line^{25} \times \mathbb{N} \times \mathbb{N}$$

The display function is designed to obey the obvious fidelity condition:

$$display(\, e_o, e_d \,) \;=\; screen, x, y$$

where

$$\begin{array}{lcl} screen & = & text[\, offset, \cdots, offset + 24 \,] \\ x & = & line - offset + 1 \\ y & = & col \\ text, line, col & = & e_o \\ offset & = & e_d \end{array}$$

However, if the definition of *screen* is to make sense, and if the cursor is to remain within the display bounds, then we must have the relevant static invariant:

$$\begin{array}{l} invariant_{static}(\, e_d, e_o \,) \;\hat{=} \\ \qquad line - offset + 1 \;\in\; \{1, \ldots, 25\} \;\; \textbf{and} \;\; offset \;>\; 0 \end{array}$$

In addition, we might want to demand display inertia, that is:

$$\begin{array}{l} invariant_{dynamic}(\, e_d, e_o, e_d{}' \,) \;\hat{=} \\ \qquad invariant_{static}(\, e_d, e_o \,) \;\Rightarrow\; e_d{}' = e_d \end{array}$$

Many adjustment functions would satisfy this: for instance, we could use:

$$adjust(\, e_d, e_o \,) \;\hat{=}\; e_d{}'$$

where

$$\begin{array}{l} \textbf{if} \;\; line - offset + 1 \;\in\; \{1, \ldots, 25\} \\ \qquad \textbf{then} \;\; offset' = offset \\ \qquad \textbf{else} \;\; offset' = max(\, 1, line - 13 \,) \end{array}$$

which keeps the offset fixed while the cursor is on screen, then scrolls to centre the cursor when it is not.

Similarly, we can extend the command set by the commands CENTRE, SCROLL_DOWN and SCROLL_UP, that centre the screen about the cursor, moves the frame down one line and up one line respectively.

$$doit_{display}(\text{ CENTRE}, e_d, e_o) \quad \hat{=} \quad e_d'$$

where

$$\textbf{if} \quad line > 12 \quad \textbf{then} \quad offset' = offset$$
$$\textbf{else} \quad offset' = line - 12$$

$$doit_{display}(\text{ SCROLL_DOWN}, e_d, e_o) \quad \hat{=} \quad e_d'$$

where

$$\textbf{if} \quad offset > 1 \quad \textbf{and} \quad line - (offset - 1) + 1 \in \{1, \dots, 25\}$$
$$\textbf{then} \quad offset' = offset - 1$$
$$\textbf{else} \quad offset' = offset$$

$$doit_{display}(\text{ SCROLL_UP}, e_d, e_o) \quad \hat{=} \quad e_d'$$

where

$$\textbf{if} \quad line - (offset - 1) + 1 \in \{1, \dots, 25\}$$
$$\textbf{then} \quad offset' = offset + 1$$
$$\textbf{else} \quad offset' = offset$$

all of which follow the "no guesses" exception principle. If we had left restoration of invariants to the adjustment function, then we would have had SCROLL_DOWN with the functionality "scroll the display frame down one line, unless the cursor is at the top whence scroll it up 13 lines". Not very acceptable (or predictable) exception behaviour!

If we want to use the "no guesses" principle uniformly for all direct display commands, then we could leave static invariant checking out of the individual command cases, leading to a subsidiary function $doit_{normal}$, and define $doit_{display}$ by:

$$doit_{display}(c, e_d, e_o) \quad \hat{=} \quad e_d''$$

where

$$\textbf{if} \quad invariant_{static}(e_d', e_o) \quad \textbf{then} \quad e_d'' = e_d'$$
$$\textbf{else} \quad e_d'' = e_d$$

and

$$e_d' = doit_{normal}(c, e_d, e_o)$$

Similarly, the dynamic invariant was of the form: "If possible without contradicting the static invariant...". We could therefore supply a "normal" dynamic invariant $invariant_{normal}$ and generate the full dynamic invariant from it:

$$\begin{aligned}
&invariant_{dynamic}(\, e_d,\, e_o,\, e_d{}' \,) \;\; \hat{=} \\
&\quad \textbf{if} \;\; \exists\, e_d{}'' \;\; \textbf{st} \;\; invariant_{normal}(\, e_d,\, e_o,\, e_d{}'' \,) \;\textbf{and}\; invariant_{static}(\, e_d{}'',\, e_o \,) \\
&\qquad\qquad \textbf{then} \;\; invariant_{normal}(\, e_d,\, e_o,\, e_d{}' \,) \\
&\qquad\qquad \textbf{else} \;\; TRUE
\end{aligned}$$

Many dynamic invariants will be able to be handled in this manner; however, some might be more complex. For instance, we might have invariants of the form: "If possible make the normal invariant hold. If not make this weaker one hold. If that's not possible make this even weaker one hold...". Of course, we could define similar generators for this form of stratified invariant.

In conclusion, this sort of interface model architecture is fairly flexible and allows the definition of many generic components to make the job of design easier. Further, it maintains a great degree of independence between the display and object at the cost of letting the display "peep" a little into the object's state occasionally. We have been able to factor most of this peeping into the generic components, making it far more acceptable. It does not cater for all interface styles, however. In particular, it is not very suitable for mouse-based systems. We will discuss these shortcomings further at the end of this chapter.

7.4 Display-mediated interaction

So far all the descriptions given have had a strong unidirectional feel: commands change the state of the system, which is reflected in the display. They do not reflect the nature of an interactive system as a feedback loop, with the user completing the loop. Although the system's response *can* be defined solely in terms of user commands, it is more faithful to the spirit of interaction to refer them to the current display. Further, there are some modes of interaction where it is impossible to separate the display cleanly from the object and retain the one-way approach. For example:

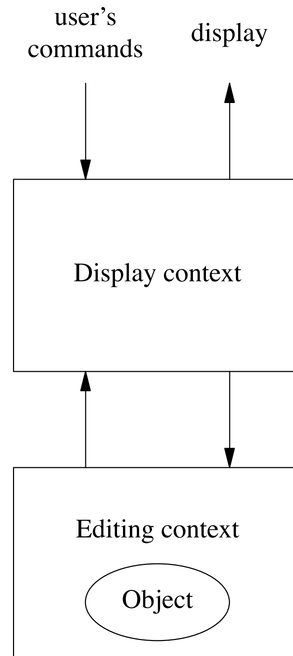
(i) Moving the cursor to the top of the display.

(ii) Moving the cursor with the screen: when a display frame moving command would mean the cursor not being on the display, the cursor is moved rather than the command failing.

(iii) Mouse input, where the meaning of location requires knowledge of display contents.

In all these cases, not only do display transitions need knowledge of the object, but the object transitions need to be aware of the display state too.

If we are to encompass such modes of interaction, our models must change to ones where the user's actions on the underlying object are mediated by information from the display context:



In terms of the model of the previous section, this would make the transitions of E_{object} dependent on $E_{display}$ as well as the other way round, effectively recombining them into a single state. Obviously we need richer models than the simple functional, data-flow-type models we have dealt with so far. These models will inevitably "open up the box" still more, and have architectural implications on the way we specify systems. Before we move on in succeeding chapters to discuss such approaches we look at one more "linear" model.

7.5 Oracles

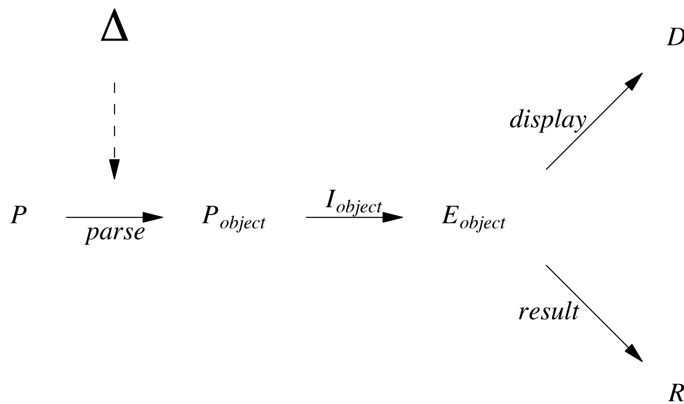
If display context is important in a system, then a detailed design will have to take account of the feedback nature of the system. However, if we are willing to relax determinism we can use a unidirectional model during the early stages of design, perhaps even during prototyping. We do this using oracles.

Consider first a system that relies on the user clicking over icons on the screen. The icons may appear at different positions, so we cannot simply map the mouse clicks onto the abstract commands denoted by the icons without using knowledge about the mapping between the internal objects and the display. In

the next chapter, we will consider controlled ways of describing this mapping, but we may not want the additional complexity at this stage. Instead we add an *input oracle* to the system that the parsing function can ask everytime it wants to know additional information about the system. So, the parsing function receives a double mouse click. It interprets the double click as an OPEN command, but then asks the oracle what lies under the mouse position. The oracle tells it that it is the folder "chapter 7" and so the parse function then translates the mouse click to the abstract command OPEN(*chapter 7*).

A similar situation might arise with a CAD system. The main palette is at a fixed screen position, so the parse function is able to translate mouse clicks into their appropriate abstract commands POINT, CIRCLE, etc. However, when the user clicks over the graphics area it would like to translate this into something like PLOT_AT(*position*). Unfortunately, it does not know about the mapping between screen coordinates and the application coordinate system. Again it asks the oracle.

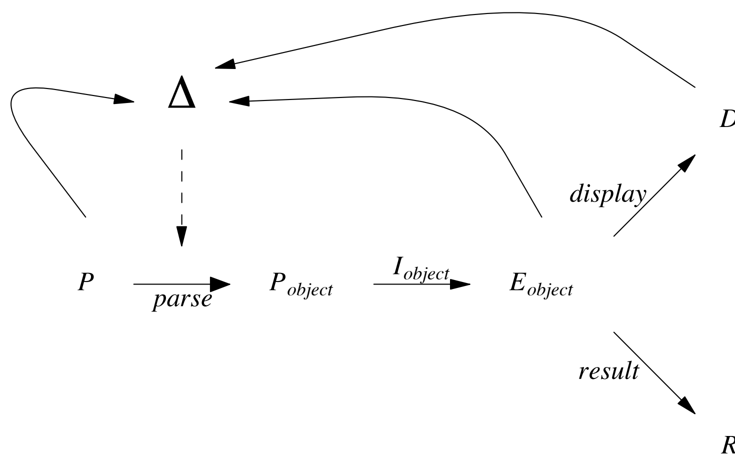
We can express this architecturally as a modified form of the PIPE architecture:



The Δ represents the information in the oracle. But where does the oracle get its information from? One approach is to treat it entirely non-deterministically. The oracle can give any reply. If we want to prove any properties of the system, we can make few assumptions about the oracle. So for instance, if some reply from the oracle would lead to an erroneous state we must assume that this might occur. We must also be pessimistic when assessing reachability or predictability properties. For a typical mouse-based system this would be very restrictive. It is equivalent to playing "pin the tail on the donkey": we are trying to get to a particular state, but our mouse clicks can land anywhere. This can be resolved somewhat if

we make some assumptions and restrictions on the oracle. When performing a reachability proof, we note any assumptions that we need to make about the oracle. For instance, in a CAD system we would want to say that the user's mouse clicks can be interpreted as being at any desired application coordinate. When the oracle gets included gradually into the system these assumptions would become assertions to be proved.

If we prefer a deterministic approach, we need to obtain the oracle's value as a function of other parts of the system. Typically it will require information from anything and everything: the complete input history, the current state, display and result, and any internal mappings defined as part of these or the parsing process. If we expressed this functionally it would give us a picture a bit like this:

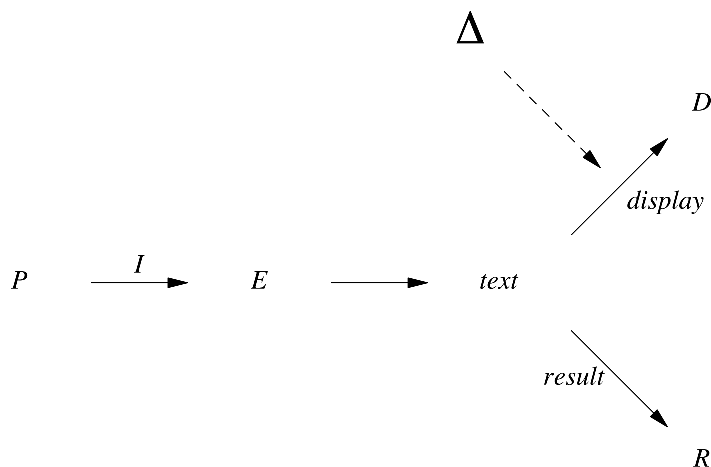


A bit of a mess! However, the whole idea of using an oracle is that the grubby parts of the design are packaged away in one area. Thus for most purposes we would use the former architectural picture, and retain a clean conceptual design during the important early stages. As the design progresses the oracle will be used less and less until eventually it disappears as part of the design. This disappearance will mean that the relevant information is obtained from the other parts of the system, but when we know exactly what information is needed the interfaces of the various parts can be modified to pass it in a clean and well-controlled fashion.

If the parsing function can be expressed in an implementable manner as a function of the user's inputs and the oracle, then the oracle can be used as part of a prototyping process. The oracle's replies could either be simulated by the designer in a "Wizard of Oz" fashion, or we could ask our favourite hacker to produce something quick and dirty to do the job. If the latter path is taken, we

can maintain a rigorous formal connection between the specification and the prototype system whilst keeping together all the kludges† that have not yet been given a formal treatment.

Oracles can be used in other parts of the system. Assume we are designing a word processor. We have defined an internal text layer with most of the complex functionality, but have not yet decided the exact rules for mapping this onto the screen. We could capture this remaining detail in an oracle, that gives, say, the text coordinate of the top left-hand corner of the display. Again, such an oracle would take information from anywhere it can. It may want to include decisions based on the type of change that has occurred and perhaps explicit user control such as PAGE-UP, PAGE-DOWN keys or an as yet unimplemented scroll bar:



Again, we could imagine using "Wizard of Oz" techniques, or quickly coding up several alternatives during a prototyping exercise. Note how the input oracle was used to cut the feedback loops, whereas this output oracle simplifies the feed-forward.

To some extent the dynamic pointer techniques in the next chapter supply in a clean, formal manner the sort of information delivered by the oracles. They thus make the use of oracles, especially input oracles, less necessary. However, as we said, we may wish to retain the simpler architecture in the early stages of design. The two techniques can work together, however, as dynamic pointer methods

† kludge – a botch or hack; something which works, but inelegantly so.

could be used to implement the oracles, or in generating "default" interfaces for use by the "Wizard" in the simulation of more complex interface styles.

7.6 Going further

We have noted how real interaction (and perhaps we ought to capitalise those two words) is often, and at its best, display-mediated. The currently displayed context is central to interpreting the user's inputs. The next two chapters present different ways of letting the display mediate the interaction. These follow roughly the two ways of viewing a layered system:

- *Editing the object* – Here we want to edit the underlying object and then have the display follow along in a consistent manner. Chapter 8 uses the notion of pointers to express this consistency. To enable the display to mediate interaction there is a well-defined way of translating commands as they pass through the display layers based on *forward* and *back* maps between pointers in the two domains. Further, to ensure fidelity between the basic objects and their display, these are separated out from the rest of the state, which is defined in terms of pointers to them.
- *Editing the display* – Chapter 9 takes the alternative view that it is the display which is dominant, and as it is the *prima facie* interface, the user is considered to be acting on it directly. It uses the notion of views. The display is a view of the underlying objects. The editor generates changes on the objects so that this view mirrors exactly the changes the user makes to the display.

Important though these later chapters are, they should not be seen as invalidating the models in this chapter. The basic abstraction in §7.2.1 is central, and all the models in the succeeding chapters are merely refinements of this. To summarise, the early discussion in this chapter told us that the relation between the functional core and the system as a whole is *not* as a *component*, but as an *abstraction*.

Furthermore, the various other models in this chapter have been shown by example to model significant systems. Even where we want some of the interaction styles that cannot be modelled using them directly, they may still be useful:

- They may be useful to model a *layer* of the full system. For instance, "move the cursor to the top of the screen" can be seen as a shorthand for a sequence of "cursor up" commands. The *parse* function would be complex, but that would not matter if we were interested simply in studying the abstraction.

- They might model a *restriction* of the system. For instance, we might remove the offending commands entirely.
- They allow modelling *non-deterministically* using oracles, which lets us temporarily ignore issues of display context during early specification and prototyping.
- Finally, they give the general *flavour* of the behaviour of more complex models, enabling us to deal better with the complexity.

This last point is important, as we are interested not just in the formal specification of interactive systems, but also in the use of formal methods to foster *understanding*