CHAPTER 11

Applying formal models

11.1 Introduction

We have seen a wide range of formal models for different circumstances and classes of systems. Formal models can be applied not only in many circumstances, but also in many different ways.

The most obvious way is as part of a rigorous formal development. The models are related to a specification and properties proved of it. The specification is then refined to a working system. The last two sections of this chapter follow this path. The first of these considers an example specification of a text editor using layered design and the dynamic pointer paradigm as proposed in Chapter 8. The full specification that this section discusses is given in Appendix II.

The last section looks at the issue of refining specifications for interactive systems. It uncovers conflicts that are are latent in the very nature of linear formal development, but are particularly important when weighed against the requirements for rapid turnover of reasonably fast-running prototypes. The section's title: "Interactive systems design and formal development are incompatible?" is a little ominous. However, the conflict arises largely because users and computers are incompatible. The job of HCI is addressing that incompatibility and the lessons from formal development are that this job is never easy.

This very rigorous approach is not the only, nor always the most useful way of applying formal models. All the way through this book we have applied the models in more informal ways. The definitions and analyses may have been formal, but the way these are related back to the real world is partly a matter of metaphor. In fact, the very nature of the formality gap means that there must always be a step between the formalisms and the things they represent. An over-

emphasis on the formal steps can make us forget this, the most important part of the development process.

There is thus a whole spectrum of ways we can use formal models, ranging from the vague use of concepts and vocabulary to the full-blown formal process described above and at the end of the chapter. So, before we move on to these more formal uses, we look at two alternative approaches. The first is a "back of the envelope" analysis of a lift system, and is similar in tone to a lot of the stories and examples that have arisen throughout this book. The second is a notation, action–effect rules, developed by Andrew Monk, a member of the Psychology Department at York. It is a deliberate attempt to link the models described earlier into a framework more digestible to the typical interface designer.

These different approaches are not exhaustive, nor are they mutually exclusive. Even where the fully formal approach is used, more informal, creative input is required. This may be independent of the models, or may be an application of the more subtle nuances of the models that are not fully captured by the formal process. Real systems tend to be like, but slightly different from, some formal model. The properties we want are close to, but with some exceptions from, the principles defined using that model.

11.2 Analysis of a lift system

Some years ago I attended an HCI conference on a university campus. The delegates were housed in several tower blocks. My room was on the 11th floor, so I often used the lifts. I am not sure whether the lifts were hard-wired or used a microcomputer, but the interface was typical of a large class of automatic systems with restricted interfaces, from electric drills to cash registers.

There were, in fact, two major interfaces to each lift. On each floor (except the basement and top floor) there were two buttons to call the lift, one if you wanted to go up and one if you wanted to go down. Now that in itself seems fairly simple, but there is often some confusion even over that. On this lift the button with the upward arrow meant "I want to go up" and the downward arrow meant "I want to go down". However, recently I was on the top floor of an hotel which had a single lift call button with an upward pointing arrow, which meant "bring the lift up". This is somewhat reminiscent of the problems with arrows for scroll boxes. Does an upward arrow mean move the window up over the text, or move the text up over the window? Fortunately, this confusion is less widespread in lift systems and the standard convention on the campus lift caused no problems.

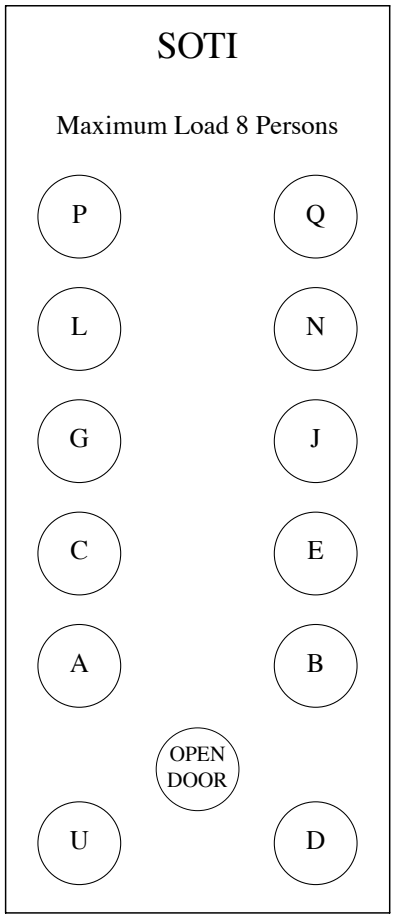


figure 11.1 *internal lift control panel*

The interface inside the lift was more complex (see *fig*. 11.1). There were a set of buttons, one for each floor. The floors were lettered and there were two lifts which served alternate floors. In addition, some letters were omitted: hence the odd labelling. Above the door was a strip with illuminated letters, which indicated the current floor, and up and down arrows indicating the lift's current direction of travel. It was this internal interface which had interesting effects.

11.2.1 The ritual

After a while I began to notice an interesting ritual I performed upon entering the lift. Other people seemed to follow the same ritual, which went something like this:

- I get into the lift, and press the button for my floor.
- I wait ...
- and wait ...
- I press the button again and at once,
  clunk – the doors close, and whoosh – up goes the lift.

When several people got into the lift the ritual became a little more complex. It began in the same way, except that buttons were pushed for each floor. The lift reached the first selected floor, one passenger got out leaving the rest of us, and then:

- We wait ...
- and wait ...
- Someone presses the button for the next floor and at once,
  clunk – the doors close, and whoosh – up goes the lift.

Now, I am sure that each of the passengers knew that the lift would have gone up anyway whether or not the button was pressed again, but we also knew that it seemed to make a difference. Analytic science versus empiricism? I am a mathematician and I did it: what chance for the experimental psychologists?

On the other hand, lifts have a reputation for unreliability and I have certainly found several times that lifts do not take me where I want them to. Computer systems share this reputation for unreliability and lack of control, though fortunately lifts rarely "crash". The compulsive pressing of the floor button is rather like the habit of repetitively striking an ENTER key.

11.2.2 Analysing the problem

Now, however well the interface had been designed the passengers may have still been impatient and engaged in compulsive behaviour, but this interface had an important flaw which is obvious as soon as the problem is described: there was no feedback from pressing the floor selection buttons. The only way the passenger could know whether the lift had registered their request was if they waited for it to go to their floor. That is rather like using a word processor with no display: you know if you have mistyped or not when page 73 gets printed. In the terms introduced earlier, the lift system is not predictable.

Once we have noticed the flaw, we can imagine simple fixes. Many lifts have a light behind each floor button, which is illuminated if that floor is due to be visited. Alternatively, if we preferred direct manipulation we could have buttons that stayed down until the lift passed the floor, but it is doubtful whether that would do much for the lift's reliability.

Analysing the problem in retrospect and classifying it in terms of generic issues is quite interesting in its own right. More importantly, we could easily have predicted the problem beforehand, using largely automatic analysis. The internal state of the lift system may be quite complex, including oil levels, maintenance indicators and the like, but the state insofar as it affects the passenger has four major components. Three of these are:

- The current location.
- The direction of travel.
- The set of floors to be visited.

If we had wanted to demonstrate the lift's predictability early in its design we would have had to show that each of these components could be obtained from the display. As the state and the system were simple, we would probably have required the simplest form of predictability, and demanded that all the relevant state was derivable from the current display. The first two of the components were permanently displayed above the door, so they were catered for. However, there was no way of observing the third component, and in this lies the heart of the problem.

I did say there was a fourth component of the state, and, to be honest, I did not even think of this component till some time after I had written out the rest. This final component is:

- The time left before the lift starts.

Now this is not just an observation of the lift, such as how long it will take to get to the third floor, which depends partly on the future of the interaction with other passengers and partly on physical factors, but is a necessary part of the state. When I get into the lift and press the button for my floor, it does not immediately start up, but instead waits for a while in case other people want to get in. Somewhere in the electronics of the lift a timer will be ticking away the seconds until the lift starts again. It is interesting, especially in the light of the discussion of dynamics in the previous chapter, that I should neglect this component, and perhaps when writing out a description of the state of a system one should be particularly careful not to miss out time dependencies.

Now, unlike the floors to visit, it did not even occur to me that we should think of displaying this component until I wrote the state out in full. Having noticed this important component, the designer might have added some feature such as an hour glass above the open-door button which gradually lost its sand until the

doors closed, or alternatively a dial, bar strip or digital indicator that ticked the seconds away. The sand (or seconds) could be topped up every time the open-door button was pressed. It is interesting that although many lifts have lights on the floor request buttons, I have never seen a delay indicator. The reader may like to imagine (as have previous readers) ways in which the passenger might be given some control over this aspect of the state.

Control panels like the lift system's are common. The underlying systems tend to have simple states. Often they are finite-state machines, or at most have a few simple additional components like strings or numbers. The displays tend to be quite simple too. There are various CAD-type packages around that help a designer layout control panels; it would be very simple to add facilities to check automatically for properties such as observability. Putting such an interface design package together with a simulator of the functionality would guard against the designer missing out components of the state, such as the door delay.

11.3 Semiformal notations – action–effect rules

There are a large number of semiformal notations used in interface design, from the psychologically oriented methods like GOMS (Card *et al*. 1983) to the systems-oriented techniques such as JSP. (Jackson 1983) Although they may contain textual descriptions or are vague in places, a significant part of these notations may be put into forms suitable for formal or automatic analyses. Unfortunately, the task-oriented techniques frequently miss the output side of a system, concentrating on the task–action mapping, which means that they are less well matched to the principles in this book than the systems description techniques.

It is important to forge a link between semiformal techniques that may be more comprehensible to a typical human factors expert or systems engineer, and the full formalisms that are capable of rigorous analysis. This section describes a notation developed by Andrew Monk, a psychologist at York, as a first attempt at bridging the gap between the types of formal model presented in this book and these semiformal notations (Monk and Dix 1987).

11.3.1 Action–effect rules

Action–effect rules consist of three elements:

- A description of the possible actions available to the user; this is equivalent to the command set of the PIE.

- A textual description or snapshot of typical displays, one for each visual "mode" or major state of the system.
- For each visual mode, a set of rules that describes how the system responds to each possible user action. This may either cause a change to the display within a mode, or cause a mode switch.

For the text editor which I am using at the moment, the possible actions would include all the keys on my keyboard, including function keys and cursor keys, plus combinations of these with various shifts. The visual modes would include:

(M1)
The start-up page with copyright notice

(M2)
The main editing mode with text covering most of the display and some context information at the top.

(M3)
Several help screens

(M4)
Setting the find/replace strings

Typical rules for the main editing screen (M2) might include:

(R1)	Printable key	⇒	character inserted in text before cursor.
(R2)	UP ARROW	⇒	**if** cursor not at the top of the screen **then** move cursor up one line on screen **else** scroll screen down one line leaving cursor at the top.
(R3)	DELETE	⇒	remove character before cursor.
(R4)	F1	⇒	**goto** Help screen (M3).

11.3.2 Analysis

These rules can be subjected to various forms of analysis. We can check them for *completeness*, and if there are any gaps in our knowledge go back and try out the various combinations. This test is fairly automatic, as we can simply see if there are any actions for which no rule is specified. We may, however, miss out some condition that would complicate a rule, making it dependent on context. For instance, rule R3 says delete the character before the cursor; however, I should really have added extra context conditions depending on whether the cursor is at the beginning of a line, or even at the beginning of the whole document. Similar problems occur with R1, and rule R2 should have yet more boundary conditions.

We can also test for *predictability*, first by checking that the modes are visually distinct, then by examining any rule with a conditional right-hand side and asking whether the user could tell from the display which of the possible responses would be taken. Designers could perform this analysis using their own professional judgement or by asking potential users. This could be done using a "live" prototype based on the description, or simply with paper mock-ups.

Consistency can be assessed by looking at the similarity (or lack of it) between rules for the same action in different modes. We would looking for several things here, most obviously, for a general consistency throughout the interface with the standard benefits of improved learning. Also, we might be especially wary where we see two modes where almost all the rules are identical, but where just one or two differ. This would be an obvious point where confusion could occur and we would have to decide whether the visual distinctiveness and semantic necessity were strong enough to overcome the propensity for error.

We can also analyse *reachability*: whether the chains of actions and modes form a completely connected net, or whether there are some blind-alley modes. This is not a totally automatic procedure, as the movement between modes may depend upon context. Also, the designer would have to analyse as a separate exercise the reachability within each mode. However, this separation of concerns would be no bad thing.

Further, we can ask about the *reversibility* of the rules: if we follow a rule, how many actions do we have to do to get back to the mode we have left? This gives us some measure of the difficulty of undoing an action. Again, this analysis would require the designer to look at the movement between modes, and also the ability to recover within (or after re-entering) a mode.

These analyses are done by eye, with the formality of named rules and modes helping the designer to book-keep. There are various subjective choices to be made by the designer during analysis. For instance, when assessing predictability, there are places where the user would not expect to know the exact form of the next display: for instance, if the user had just read in a new file. The designer must distinguish these cases from those where predictability has been violated in a serious manner. It is also partly a matter of opinion whether the effects of actions are similar in different modes. For instance, in an integrated system with word processor and spreadsheet it might not be clear what the consistent spreadsheet equivalent to the word processor's carriage return would be. The rules therefore give the designer a structure within which to perform analyses, but do not take over from the designer's professional role.

11.3.3 Layout choices

There are also choices to be made during the laying out of the rules. By introducing more visual modes the number of conditional rules can be reduced. For example, we could introduce 24 modes, one for each line the cursor could be on. We would then have a copy of rule R2 one for each of the 24 modes. Each rule would be simple, as there would be no conditional, but there would be lot of rules! In addition, the conditional would reappear as a violation of consistency. Similarly, there are several Help screens that can be scrolled through using the cursor keys. These could be regarded either as one mode with a rather vague rule:

mode M3:	(R5)	RIGHT ARROW	⇒	change Help information field to next subject.

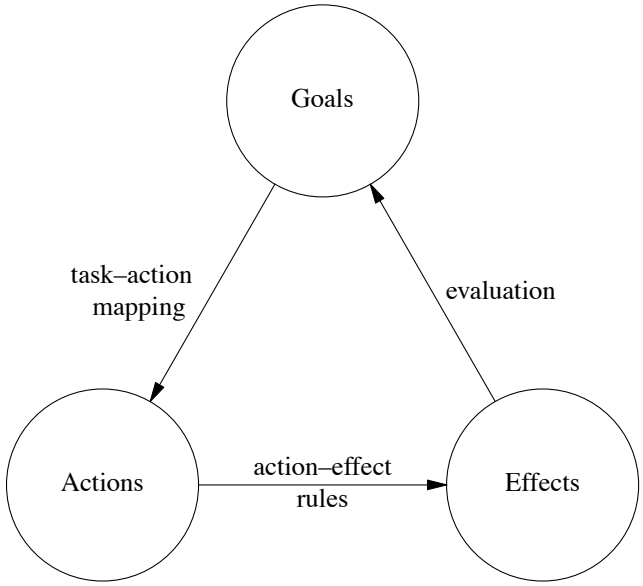
or as lots of modes with rules of the form:

mode M3a:	(R5a)	RIGHT ARROW	⇒	**goto** second help screen (M3b).
mode M3b:	(R5b)	RIGHT ARROW	⇒	**goto** third help screen (M3c).
...			...	
mode M3i:	(R5i)	RIGHT ARROW	⇒	**goto** first help screen (M3a).

This ambiguity could be addressed by extending the rule system to include some sort of inheritance hierarchy, but this would significantly complicate the notation. The designer can also choose to be more or less precise about the descriptions used. Obviously, a very vague rule will not be very useful for analysis.

11.3.4 Comparison with task–action mappings

Most of the psychologically founded notations are task-oriented: the focus is on how the user's internal goals are translated into actions upon the system. Typical of these would be GOMS (Card *et al.* 1983) or TAG. (Payne 1984) These are, in effect, designer's models of the user's actions. We shall refer to the whole genre as task–action mappings during the following discussion. Action–effect rules look at a different part of the user-system feedback loop:



The final side of the loop is labeled *evaluation*, from Norman's analysis of interaction. (cd) It is a pity that psychologically founded notations are poor on this side, as this would nicely complete the picture.

It is clear that task–action mappings and action–effect rules occupy different design niches and can therefore complement each other in the design process. To get an idea of when we should be using one approach rather than the other, we shall contrast their respective strengths and weaknesses.

The task–action approach seems at first glance more user-oriented; the action–effect rules merely describe the system. They do, however, describe the system from the *user's* perspective. We could contrast the two approaches as looking *inside* the user, compared with standing *alongside*.

Concentrating on task is clearly important in optimising systems for typical behaviour. It enables the designer to match the form of the dialogue with the structure of the tasks the user will perform. The danger of this approach is that, by optimising for particular tasks, a system may deal badly with those that are missed. New systems are always used in ways that the designers did not consider. Good task analysis may help to predict these, but will never discover all possible uses. In addition, we have to consider error recovery, which massively increases the set of "tasks" that should be allowed for.

Action–effect rules have completely opposite strengths and weaknesses. They favour no particular task and, without being paired with a task-oriented input, would allow systems that were totally unsuited to their eventual use. However, by being uncommitted they favour systems that can be used in new unforeseen ways. In particular, by focusing on the effects that a user can achieve and how to achieve them, it allows users to set their own goals.

A similar contrast occurs in descriptions of direct manipulation. On the one hand, there is the concept of cognitive directness (Hutchins *et al.* 1986), the natural flow of user's mental models and goals into the system. On the other, is the concentration on exploration, recognition and goal seeking. (Shneiderman 1982) The former gives us the clues to what metaphors are relevant within a particular system, but does seem to assume a blind and deaf user. The characteristic user of the latter would, I suppose, be extremely sensually acute, but with little foresight.

11.3.5 Prototyping and automatic analysis

Some of the rules given above were very explicit, in particular those that sent the system into new modes. Others were almost entirely textual. With a little effort the major mode changes could be captured entirely, and the notation rendered in a machine-readable and implementable form. This would allow various levels of automatic design support.

First, it would form a rather crude prototype. The resulting system would behave in a gross manner like the target, the user's inputs taking the prototype through its visual modes. More detailed behaviour might be met with a dialogue box saying "the character X is inserted before the cursor", but with no actual change to the display. This would be rather similar to the slide-show prototypes built using hypertext systems.

The same description could be used for some automatic checking. Some of the analysis presupposes a level of interpretation, but some of the more tedious checking and book-keeping could be taken over by the computer. Further, when a detailed specification is produced this could be compared with the action–effect description, and checked for consistency.

11.4 Specifying an editor using dynamic pointers

In Appendix II a specification is given of a simple editor. This section gives an overview of this example specification. It is a simplification of an editor specified and implemented as part of an experimental hypertext system at York several years ago. The original included a folding scheme that allowed disparate parts of a document to be viewed at the same time, a more sophisticated pretty-printing mechanism and hypertext links to other documents. The editor sat within individual windows of a multi-windowed system including a structural view of the hypertext.

The specification is an example of a layered design. It is specified in three layers:

- *Display* – The physical display as seen by the user. It represents a standard 80 by 25 character screen.
- *Text* – A formatted representation of the application objects, but of unbounded extent.
- *Strings* – The application object itself, a simple stream of characters.

Each layer has an associated pointer space, and the layers are linked using projections (*fig*. 11.2).

The specification proceeds in two major stages. First, the major data types are defined, *Disp*, *Text* and *String*, these being related by appropriate projections. These data types are then used to define states, with appropriate projection and interaction states being added to the various layers.

Proceeding in this two-step fashion has various advantages. The operations defined over the basic data types are "pure": that is, there is no concept of update, new values are merely constructed from old. It is often argued (ce) that it is easier to reason about pure functions. Thus we can perform some proofs or analysis of the specification at this level.

The operations will typically not correspond directly to simple keystrokes. For instance, the insertion operation on a string requires both a character and a position at which to insert it. Similarly, a find/replace command requires two strings as parameters. These additional parameters are part of the "interaction state" of the system. The former would typically be an insertion point, and the latter may reside as some permenant part of the state of the system or be demanded interactively as part of a find/replace dialogue. These decisions can be left to the second phase of the specification, thus simplifying both stages.

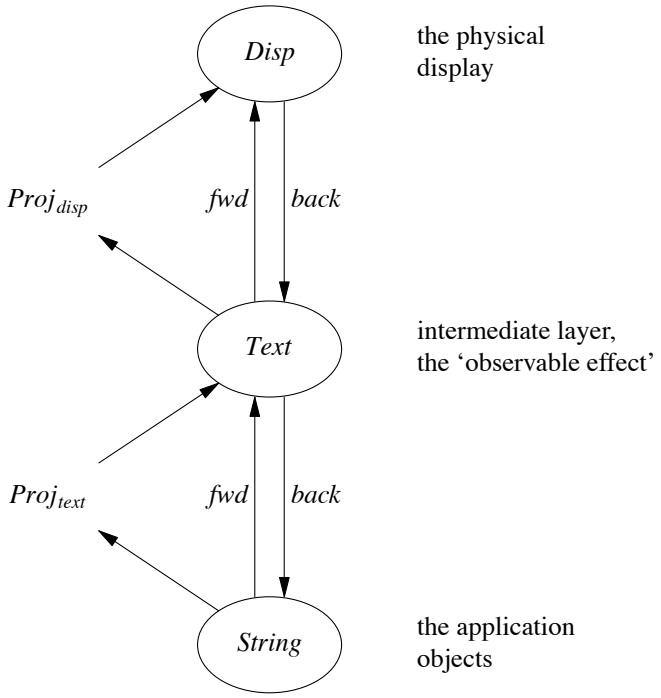


figure 11.2 *pointer spaces and their projections*

The role of the projection distinguishes the specification given here from similar specifications given elsewhere (Ehrig and Mahr 1985, Chi 1985, Sufrin 1982). The information in the projection would typically be part of the state definition phase and would be likely to be dispersed throughout the specification. The use of pointer spaces means that this information is both separate from the data types themselves, and part of the functional data-type phase. This both eases tractablity and factors the design.

11.4.1 Data types and projections

The display and text are very simple, the display being merely an 80 by 25 array of characters and the text a sequence of lines. The pointers for each of these are merely number pairs giving row–column offsets. The projection between *Text* and *Disp* is just a window into the text at a position specified by a text pointer. The *fwd* and *back* maps associated with this projection are simple enough in the "normal" case; one merely adds or subtracts the relevant offset. There are, however, numerous special cases. For example, when we wish to take

a display pointer *back* to a text pointer, if the display pointer is over blank space we want it to be mapped to the "last" sensible text position. We cannot simply say that such mappings are out of range, as we will want to translate mouse clicks after the end of lines and treat them as selecting the appropriate line end. The important point to note is that these special cases are isolated in the description of the *back* and *fwd* mappings, rather than being spread throughout the definition of each operation on the objects.

The actual objects being manipulated are simple streams of characters including new-line characters. The pointers to these are single integer offsets into the strings. The string data type has two operations defined which yield new strings and pull functions:

$$\begin{array}{ll} \mathit{insert}: & P_{string} \times Char \times String \rightarrow String \times (P_{string} \rightarrow P_{string}) \\ \mathit{delete}: & P_{string} \times String \rightarrow String \times (P_{string} \rightarrow P_{string}) \end{array}$$

These are simply character insertion and deletion at the appropriate positions. In addition, there are *succ* and *pred* functions which take a pointer and give the succeeding or preceding pointer. Each of these requires not only the pointer but also the string on which it is acting. This is so that the operation can do "range checking" and thus yield only valid pointers.

The projection between strings and text is merely the breaking up of the string at new-line characters. The line breaks are defined using a sequence of string pointers, *line_starts*, which give a pointer to the beginning of each line. Thus *line_starts*(3) is a pointer to the beginning of the third line. A more sophisticated approach would have been to have block pointers encompassing each line, but would be essentially similar.

Allthough not part of the example, it is possible to update this information in a highly efficient manner using locality information (§8.4). If we know that only a small part of the underlying string is affected by an operation, then we can simply use the *pull* function to modify parts of this *line_starts* structure outside the locality. This seems a little over the top for such a simple projection function, but forms the pattern for more sophisticated treatments. In the full version of the system a more complicated pretty-printing scheme was used. (cf) With the addition of contexts for the projection as described in §8.4.7, the structure of the system was very similar.

Although the action of splitting a string at line breaks is very simple, the *fwd* and *back* functions require a surprising amount of care at the boundary cases. As with the text–display projection this area of potential mistakes is localised to the definition of the projection, rather than being spread around the specification. It is thus more likely to be both correct and consistent.

11.4.2 String state

The lowest level of state in the system is the stream of characters being manipulated. It is just the current value of the string being manipulated, and thus the "state" is the same as the data type. This is so simple that it is not defined as a state as such in the specification, but is merely a component of the text state. The operations on strings as a data type can be regarded as operations on the underlying string state. The character required for the *insert* operation is likely to come directly from the keyboard and will be passed in as part of the user's command. The point at which to put it and the point required by the *delete* operation will be the cursor or insertion point. This will not be supplied by the user on a character-by-character basis, but will be part of the interaction state of the system. This is the additional information required to form the next level of the state of the system.

11.4.3 Text state

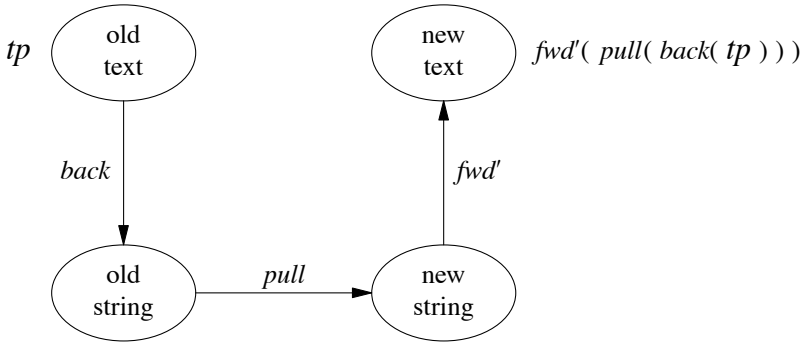
The state at the text level consists of a string (the lower level of state) plus a string pointer which will be the insertion point. The text is obtained by using the string-to-text projection. This projection has no parameter information and thus nothing additional is needed in the text-level state. Operations on this level of the state are of three classes:

Update – *insert*(*c*) and *delete* are inherited from the string layer and "passed" on after suitable conversion.

Movement – *move_right* and *move_left* are operations which move the insertion point. These use the *succ* and *pred* operations at the string level to update the insertion-point component of the state, and hence automatically ensure that the pointer is valid.

Selection – *select*(*tp*) is an entirely new operation which sets the current insertion point to the text pointer *tp*.

The update operations are very similar to one another: they simply add the insertion point as an extra parameter and update the string part of the state using the appropriate string operations. The new insertion point is obtained by applying the *pull* function from the string operation. The display level will want an equivalent of the *pull* function on the text pointers in order to update its own state. This is obtained in a canonical way, by first translating text pointers to string pointers using the *back* function on the original string. The string pointer can then be updated using the *pull* function before being translated back to a text pointer using the *fwd* pointer of the updated string:



This construction is the same no matter how complicated the projection, and forms part of the pointer-space "tool-kit" for constructing manipulative systems.

The update operations are very simple, applying *succ* or *pred* to the insertion-point component of the state. Selection is almost as simple, it sets the insertion point to the supplied text pointer. However, as the insertion point is held as a string pointer, it must first be translated with the *back* function. An alternative design would have the insertion point as a text pointer. The cursor would then be able to range all over the semi-infinite text space, and not be constrained to the formatted region. The *back* function would then have been applied before passing the insertion point to the update operations.

The text state has a "display" mapping $display_{text}$ which uses the projection to obtain a text and translate the insertion point to a text pointer. This is not a physical display, but would correspond to the virtual display that most users would perceive as lying behind their screens. The result mapping of the system would be either the actual string or the text portion of the projection. The former would be the case for a text editor in a filing system, the latter for a printed document.

11.4.4 Display-level state

At the physical level, the user sees a screen and presses actual keys. The projection from text to display requires a text pointer as a parameter. This text pointer is added to the text state to obtain the complete system state. This can be though of as the display "map" component of the state.

The commands at this level are all inherited directly from the text level. Update commands are simply the appropriate keystrokes, as would be the movement commands. The selection command would be obtained from a suitable pointer and thus be a display pointer.

Selection is the simplest operation, which is not surprising as the whole dynamic-pointer paradigm revolves around the issue. The display pointer is simply translated into a text pointer using the *back* function from the text-to-display projection, and the selection operation passed on to the text level. The text level will, of course, go on to translate the pointer yet again! However, this does not concern the display level.

The esential structure of update and movement is quite simple. The relevant operations are performed on the text state and, in the case of update, the text pointer "map" component is updated using the *pull* function. However, both update and movement are complicated by the existence of a *static invariant*.

A standard observability condition is that the insertion point be visible whenever operations like insert and delete are performed. There are several ways of accomplishing this:

- The operations can be made illegal when the insertion point is not visible.
- The insertion point can be scrolled into view when an update operation is performed.
- The insertion point can always be kept on screen.

I will not go into the various arguments now, but the specification chooses the last option. This then becomes an invariant of the system. It can be expressed by demanding that the insertion point (treated as a text pointer) is obtainable by applying *back* to a display pointer. Because it relates the state and display at each instant, it is a static invariant.

As we discussed in Chapter 7, it is usually the case that to maintain static invariants some sort of adjustment must occassionally be made to the display mapping. An *adjust* function is defined in the specification for this purpose. It takes an updated text state and the *pull*ed map component. If the pair satisfy the static invariant they become the new display state. This is a simple form of display inertia. If, however, the new insertion point lies outside the display, a new text pointer is chosen for the map component. The choice of this position includes various cases depending on where the insertion point lies relative to the beginning and end of the text, but essentially centres it in the screen. Again, a lot of complexity and boundary issues that might otherwise be distributed about the system are collected within the *adjust* function. As with the projection functions, this both simplifies the rest of the system specification and ensures consistency.

The actual display of the system is obtained by using the text-to-display projection and using the *fwd* function to take the text-level insertion point into a display pointer. The final display then consists of the character map together with a cursor position. No new commands were added at the display level, but commands such as scroll-up or scroll-down could be be added here, updating the map component appropriately.

11.4.5 Summary architecture – adding new functionality

In retrospect, we can look back on the state architecture and see how commands are passed consistently back from level to level, and the display obtained (*fig*. 11.3):

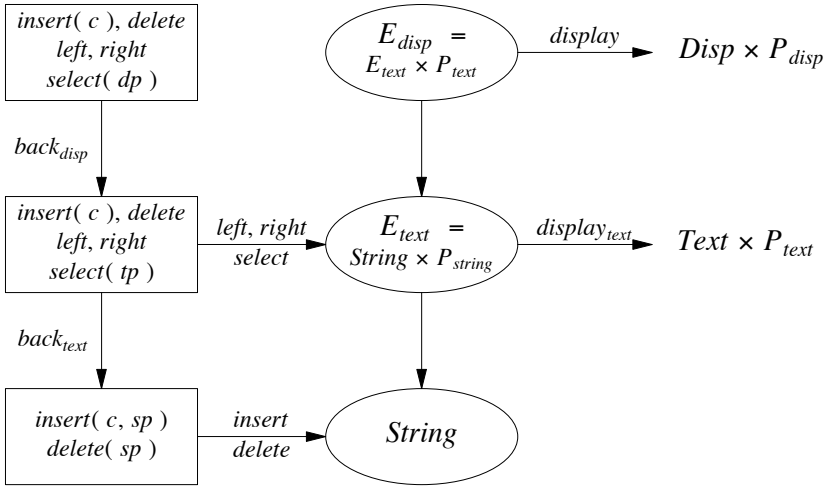


figure 11.3 *summary architecture*

The *back* arrows on the left-hand side show how the user-level display pointers are translated back through text pointers and, where appropriate, to string pointers. Of course, this is not the only aspect of translation. Pointers may be added (as with the insertion point to the update operations), or other data types. Also, the translation scheme may be far less one-to-one than this simple system.

The arrows down from E_{disp} to E_{text} and from E_{text} to *String* remind us that they are related by an abstraction relation. Although the dynamic-pointer paradigm has allowed us to build the various layers from one another, the essential feature is still one of abstraction.

A lot of the complexity of the system is wrapped up in the definition of the projection mappings and the display adjustment function. New commands are quite easy to add. As an example of this, the Appendix shows how a find/replace command would be added. We assume we have such a command for strings. It takes two parameters, the search string and its replacement, and updates the string returning also a *pull* function. It is, of course, this pull function which

then makes the rest of the process easy. The find/replace command is plugged in in exactly the same way as the other update commands. The text layer simply updates its insertion point using the *pull* function, and produces a *pull* function for text pointers by using the *back–pull–fwd* trick. The display level again updates its mapping component, a text pointer, using this pull function, then applies the *adjust* function. The only extra mechanism that is required is choosing somewhere to add the string parameters. For a self-contained editor the appropriate place would probably be at the text level, with commands to select editing of text or these strings. In a more integrated environment these may be supplied as part of a high-level dialogue.

The mechanisms described above might seem a trifle heavy for a simple text editor. However, they factor nicely even this design. More importantly, the basic architecture using pointer spaces is the same for more complex systems. A general design strategy using dynamic pointers goes something like this:

- Define application objects and add pointers for them.
- Choose appropropriate user-level layers corresponding to the physical screen and the "observable effect" and add pointers to these (the screen pointers will be character or pixel coordinates).
- Generate the projections between these and the application objects.
- For each operation on the application objects, add a *pull* function.

The rest fits effortlessly together... well almost!

11.5 Interactive systems design and formal development are incompatible?

The most rigorous use of formal models would be as part of a strict formal refinement process. However, when we consider the relationship between the requirements of the *process* of interactive system design, and the *process* of formal development, several conflicts occur. These problems are not unique to interactive systems design, but are inherent in the concept of formal development; it is just that the rigours of interactive systems intensify and bring these problems to light.

After considering the differing requirements of the two domains of interactive systems design and formal refinement, we will proceed in a dialectic style. Conflicts will become apparent between the sets of requirements, which we will attempt to resolve, eventually leading to the need for structural transformation at the module level during refinement.

We will find that although interactive systems design and formal development are not completely incompatible, their combination is both complex and instructive.

11.5.1 The requirements of interactive systems design

Interactive systems design gives rise to three problems:

- Formality gap.
- Rapid turnaround – iterative design.
- Fast prototypes.

These are described below:

Formality gap

We discussed this in the first chapter. This is the gap between the informal requirements for a system in the designer's or client's head, and their first formal statement. The steps within the formal plane are relatively simple compared to this big step between the informal and the formal. If the formal statement of requirements is wrong, then no amount of formal manipulation will make it right. Correctness in the formal domain means preserving the badness of the first formal statement.

We have noted that if an abstract model is designed well for a particular class of principles, it can help bridge the formality gap. To achieve this, there must be a close correspondence of structure between the abstract model and the informal concepts. However, the abstract model will capture only some of the requirements, and the same principle of *structural correlation* must then apply to the entire interface specification.

Rapid turnaround

Because of the formality gap, we will never entirely capture the requirements for an interactive system (Monk *et al*. 1988) and thus some form of iterative design cycle is usually suggested. That is, a prototype system is built based on a first guess at the interface requirements; this is then evaluated, and new requirements are formed. The turnaround of prototypes must be fast for this process to be effective, perhaps days or even hours. Frequently this is done using mock-ups that have the immediate appearance of the finished product, but lack the internal functionality. However, many of the interface properties we have studied permeate the whole design of the system, and hence we would see it as necessary to have some reasonable proportion of the functionality in this prototype.

Fast prototypes

Not only does the turnaround of prototypes have to be rapid, but the prototypes themselves must execute reasonably fast to be usable. A slow interface has a very different feel from a fast one, and hence wrong decisions can be made if the pace of the interaction is unreal. The evaluator may be able to make some allowance, but this ability is limited. It would be very hard to evaluate an interactive system where you type for a few seconds, then have to go away for ten minutes and have a cup of tea before seeing what you have typed appear. The only thing to be said in its favour, is that it might encourage predictability, as a result of the "gone for a cup of tea" problem! Not only is it difficult to appreciate the system at all, but also very poor performance can encourage the wrong decisions to be taken, negating the benefits of prototyping. For example, imagine designing a word processor. One design is a full-screen, "what you see is what you get" editor, the other a line editor with cryptic single-character commands. At the speed envisaged in the production version, the full-screen editor would be preferable, but when executed a hundred times slower, the line editor, requiring fewer keystrokes and not relying so much on screen feedback, would appear better. It was for just such situations that line editors were developed! Of course, such a major shift would be obvious for the designer; however, there may be many more subtle decisions wrongly taken because of poor performance.

Contrast with non-interactive systems

We can contrast the above requirements with the design of non-interactive systems such as data-processing or numerical applications:

Requirements contrast	
Interactive systems	DP or numerical applications
wide formality gap	well-understood requirements
rapid turnaround – iterative design	slow turnaround
fast prototypes – for usability	functionality sufficient

Whereas the requirements of interactive systems are very difficult to formalise, for a DP application like a payroll, this may not be too much of a problem. The requirements are already in a semiformal form (e.g. pay scales, tax laws) and they are inherently formalisible. Similarly, because these requirements are well understood, there is less need for iterative design: changes in requirements may be over periods of months or years, with corresponding turnaround times. Finally, with such applications, it is sufficient to prototype the functionality only, with little regard for speed. It is easy to produce a set of test data and then run a numerical algorithm overnight to check in the morning for correctness. Even complex distributed systems may be able to be simulated at a slow pace.

11.5.2 Formal and classical development compared

In this section I compare formal development with classical development. By classical I mean a non-formal approach to development (hacker rather than Homer). I assume that the requirements are known and consider how these requirements are used to produce a first implementation, which is then optimised to create a version that is suitable for use (perhaps as a prototype or even as a product). Finally, I examine how these two processes respond to changing requirements, and the implications this has for the development process.

Initial development

Classical

Given loosely stated requirements, the programmer will, possibly using some intermediate graphical or textual plan, produce a first implementation of the program. Typically this will already be structured with efficiency in mind. However, if this is not fast enough, several stages of optimisation may ensue before reaching a version suitable for release (*fig*. 11.4):

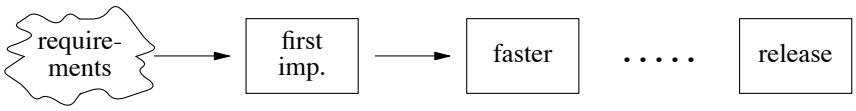


figure 11.4 *classical development*

Interspersed with this optimisation process will be debugging. In principle, one might debug each version in turn until one is sure of correctness. In practice, this debugging will be distributed, errors present from early implementations being corrected only later in the process. These changes will probably never be reflected in those early points, as the early versions are likely to be overwritten, or at best stored in a source control database.

Formal

At this stage the formal development process is quite similar. The requirements will be used first to generate a specification. This first specification will then go through several levels of refinement within the formal notation, both to make it sufficiently constructive for implementation and possibly as a first stage in moving towards an efficient formulation. The final product of these specifications will be used to derive the first implementation, which itself may then be optimised through several versions before a releasable version is produced (*fig*. 11.5):

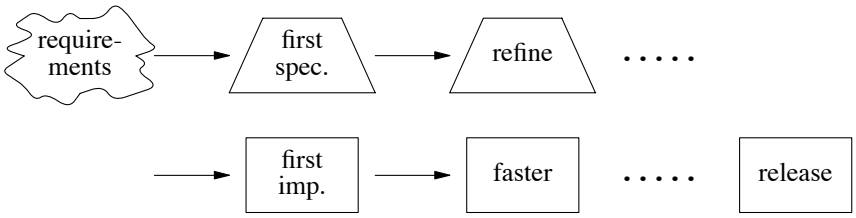


figure 11.5 *formal development*

Again there will be debugging steps, both in the specification as each refinement is checked against the previous specification for correctness, and in the implementation as this also is checked for correctness. The only difference here from classical development is that debugging is less likely to involve errors from previous versions. In particular, because the initial specification is derived much more directly from the requirements (it may in fact be a formal statement of the requirements), there are likely to be less changes needed to make the final product match the informal requirements. With some formal development paradigms, such as refinement by transformation, it could be argued that there is *never* any debugging, as all versions are guaranteed to be correct. However, even here backtracking in the transformation process is a form of debugging. From now on we will largely ignore these debugging steps, as they form a development process at a finer granularity than we are considering.

Changing requirements

Classical

If the changes in requirements are extreme, the programmer may be tempted to throw away all previous work and start from scratch. More usually, however, the most developed version of the system will be used and altered to fit the new needs (*fig*. 11.6). Because the fastest, optimised version is used there is unlikely to be much need for optimisation steps, except where radically new algorithms have been introduced. Note especially that the changing requirements are not seen to affect at all the early unoptimised versions of the program; they are just history.

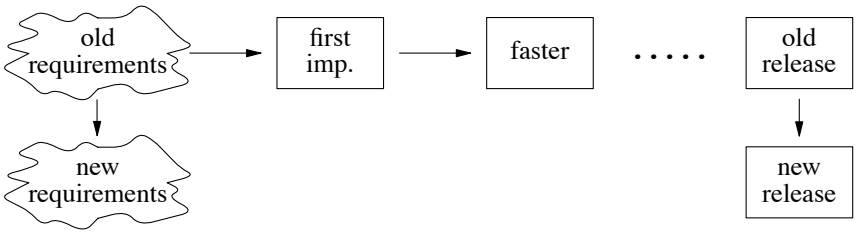


figure 11.6 *classical development, change in requirements*

Formal

The formal situation is very different. The intermediate versions, and especially the first specification, are the *proof* that the final version really does satisfy the requirements. There may well be testing and validation as well, but it is the process of development itself which is the major source of confidence in correctness. This is even (and especially) true when the process does not employ automatic checking. It is insufficient (although tempting) to change the specification a bit, change the final version a bit, and say the process is still formal. No, for formal correctness a change in requirements demands a complete rework of the whole development process, from initial specification to final optimised system (*fig*. 11.7). Again, in the worst case, this may involve a complete rewrite, but will usually be obtained by propagating small changes through the stages:

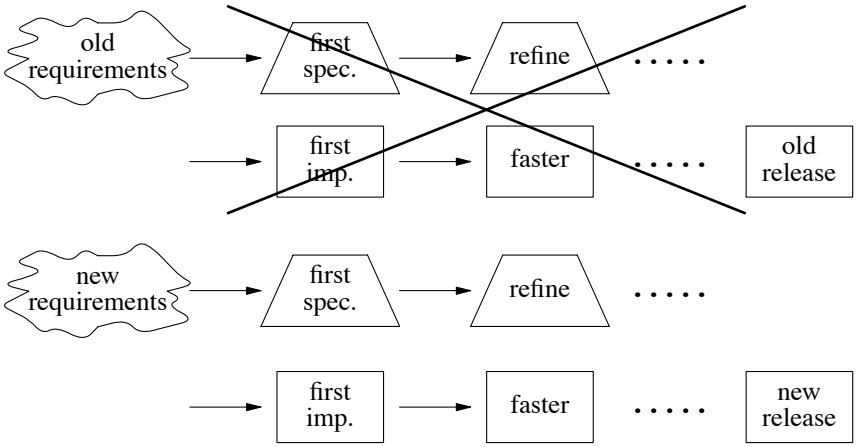


figure 11.7 *formal development, change in requirements*

11.5.3 Conflict – rapid turnaround and refinement

How do the requirements for interactive systems design fit into this picture?

- Rapid turnaround ⇒ lots of changes in requirements, say m in all.
- Fast prototypes ⇒ lots of steps in the refinement process, say n steps.

That is, the length (n) of the development chains, and the number of them (m), are far greater for interactive systems compared with non-interactive. These interact with the cost figures:

- *Classical* – This takes n steps to produce the first version, and one further step for each requirements change, so $m + n$ versions will have finally been produced.
- *Formal* – This too takes n steps to produce the first version, but then all requirements changes also require n bits of work, so the number of separate specifications and programs eventually produced will be $m \times n$.

At first glance the difference between these two cost figures, $m + n$ for classical development and $m \times n$ for formal development, is astounding. One wonders if formal development can ever be a practical proposition. A one-off cost can be acceptable, even if high, but a recurring cost like this could never be.

Happily, the situation is not as bad as it seems. First, the number of "changes in requirements" may be vastly different. For a system using classical methods, many of the requests coming in for changes will not be true changes in requirements, but restatements of parts of the original requirements the system does not satisfy: that is, long-term debugging. As we've noted, formal

development should reduce this to a minimum (or even zero if the requirements are a formal document themselves). Unfortunately, this argument does not hold too well for interactive systems design, as we've noted that here formal methods will not perfectly capture informal requirements. Again, the number of development steps may differ. Formal development may require the additional refinement steps, but the costs of each step will be much reduced, as there will be less debugging.

Orthogonal development

It is clearly critical just how costly the changes to the intermediate specifications are when requirements change. In the best case, the n changes needed for formal development may be a distribution of the effort of the single change in the final implementation for classical development. We would hope that a small change in requirements, would only require a small change in each of the intermediate specifications.

Modularisation is the standard vehicle for such localisation. A small change in requirements is likely to require changes in only one or two modules of the specification. If the refinement and the implementation preserve the specification's module structure, then we can alter the relevant modules in the refined specifications and implementation and re-use all other modules. That is, the refinement process must obey the principle of *structural correlation*.

Cedar (Donahue 1985) uses such a system, carefully maintaining dependency information so that recompilation upon the change in a module is minimised. (Compilation is, of course, a form of refinement.) However, the idea of locality in Cedar, as in most traditional languages such as Ada (Ichbiah *et al*. 1983), is based on type correctness. The semantic repercussions of changes are not considered, and it is not obvious therefore that the rest of the system will behave as expected.

Happily, the situation in formal development can be much better. Modularisation based upon semantically defined interfaces leads to full semantic independence between modules. One module cannot affect another module without a change in the interface. This leads to an *orthogonal* development idiom. The specification consists of many modules, as do each of the intermediate stages to the final implementation. Development within any one of those modules is independent of all the rest. If a change is made in one specification module then an early, perhaps very inefficient, version of the corresponding implemented module can be used with highly optimised existing modules. This means that we can get a reasonably fast prototype back into the hands of the interface designer very rapidly.

11.5.4 Conflict – structural correlation and orthogonal development

To summarise where we have got to: the formality gap forces us to match the structure of interaction in the original specification. The conflict between the need for rapid turnaround and fast prototypes, and formal refinement forced us to use orthogonal development, with structural correlation throughout the refinement process. Thus we will arrive at an implementation with a structure still matching closely the structure of interaction. However, at the end of the day, in order to obtain a fast prototype we will need an efficient implementation structure. Unfortunately, the interaction structure and the structure required for efficiency do not usually agree.

It is worth noting that the experience of formal development is likely to be very different here from classical development. In the latter, the structural design will from the beginning be oriented towards efficient implementation. In fact, the same can be done with formal specifications. However, we argue that this is not what a specification is for, and such trends are likely to yield errors in translating requirements to specification. (Jones 1980) In short, a good formal specification is likely to have problems with orthogonal development.

Non-orthogonal development – interface drift

However, despite large gains through intra-module optimisation, the time may come when the system is still too slow and inter-module optimisation is needed. Design steps which involve several modules are *non-orthogonal*, since subsequent changes to the specification of one module will require the other modules to be redone from the non-orthogonal design step onwards. The challenge is therefore to achieve this in as well-structured a way as possible, preserving correctness, and making easy the re-use of existing modules as requirements change.

The simplest situation involves just two modules. We can think of one as a *server* providing some functionality, and the other a *client* which uses this service. The need for intra-module refinement shows up in two ways:

- *Information* – The client module could perform further optimisation if the server could give it some more information: for instance, the use of locality information to tell the client about where the server has made changes. This locality information is not usually present in the original specification interface and is therefore not normally available to the client, although the server "knows" it. This type of information may enable the client to optimise changes in its own data structures.
- *Services* – There may be computations in the client module involving many calls across the interface to the server. Such operations may well be performed far more efficiently by the server, reducing function call overhead at very least, but quite likely being more efficient in general due to the server's knowledge of representation and (again) information not available across the interface.

As we can see, the two are not entirely independent, and both require *interface drift,* a movement of some information or functionality across the interface between two modules. In general, the information category requires a drift of functionality across the interface *upwards* from server to client, as the server "opens itself up". Similarly, the services category requires a *downwards* drift of functionality, as the server "takes over" jobs previously performed by the client.

A technique for managing the latter form of interface drift was developed at York as we struggled with the refinement of an interactive system (Dix and Harrison 1989). This technique involves the encapsulation of the functionality that is to be transferred into a separate module. The functionality is then transferred in two steps. The first separates the encapsulated functionality from the client module. The second step then merges this with the server. The existence of the additional module both simplifies proofs and also allows substantial independent development of the two modules despite the non-orthogonal step.

This technique for interface drift has been cast into a general framework for formal manipulation of specifications at the module level (Dix 1989). Important issues that arise when non-orthogonal steps are taken include the ability to trace when they occur and maintaining control if the process involves more than one person. Interestingly, it is the formal *interface* between the modules that is crucial in the approach and, in particular, promoting it to a "first-class" object in the development environment.

11.5.5 Human interfaces for development environments

The discussion has focused on the rigours placed on formal development when the subject of that development is an interactive system. However, it leads to important questions about the cognitive demands of the resulting process, and the user interface to any support system. Recalling the comparison between classical and formal development processes, the classical took $m + n$ steps to the formal's $m \times n$. Although we argued that the actual effort expended would not be as extreme as this suggests, it is likely that the number of *documents* (formal and informal) produced will approach this level. That is, the complexity of formal specification "in the small" is likely to approach that of classical programming "in the large".

Large-scale programming has in the past been supported by well-documented analysis and management structures, and currently project support environments are being developed to aid this process. However, the timescales involved in interactive systems design preclude these approaches, requiring instead environments more akin to exploratory programming. (Goldberg 1984) Marrying these conflicting styles will require exceptional organisation and ingenuity in the environment and its user interface.

Whilst the development process is totally orthogonal, the complexity is unlikely to be too bad as the documents can be located in a matrix, the dimensions of which are reasonably small. However, non-orthogonal development steps significantly complicate this picture. They make the structure more complex, introducing problems of naming and representation. Whilst such issues are apparently insignificant formally, they have a major effect on the usability of formal development environments.

11.6 Summary

This chapter started by placing the various sections on a gradation based on formality. We began with the least formal and moved on to more and more rigorous approaches. However, there is a different way of classifying the examples based on *professional* skills.

All the examples required different skills and expert input of various kinds. Let us begin with the full formal refinement process (§11.5). The specification, which is the starting point, captures the user interface requirements (and most other requirements); hence there is little if any room for further human factors input. This is an advantage, as we need only look for someone with software engineering skills to manage this part of the design process. If we use the analogy to the building trade, this stage corresponds to working out the order and method of construction given detailed plans. The building may not take shape in

the order it was drawn, but it will end up as the architect planned it. (Well almost: in both interface and construction there is likely to be some feedback.)

The production of a formal specification of a specific system (§11.4) will inevitably involve design decisions which affect the interface. Thus the practitioner will require skills both in formal methods and in human factors (or at least, have ready access to appropriate advice). This may well be a two-stage process, starting out with a well-fleshed-out but not rigorous specification of the interface produced by someone with human factors expertise and a smattering of the appropriate formal knowledge, which is then continued by someone with complementary skills. If we look at the use of the formal models in this stage, the important activity is selecting the appropriate models and interface architectures to use and then applying them. We will discuss this selection process in more detail in the next chapter. Once the appropriate models are chosen their application is more or less mechanical. The building equivalent would be the drawing up of detailed plans, using formal techniques such as stress analysis where appropriate. The skills required are draughting, understanding of human features in the building not captured by formal analyses, and knowing just where the formal analyses are appropriate.

The action–effect rules (§11.3) capture some of the properties of formal models in a less formal (and perhaps less frightening) framework. The appeal is to someone with reasonable human factors skills but little if any formal expertise. Certainly, no knowledge of formal modelling as expounded in this book is required over and above that within the framework itself. Thus the formal models are not applied directly but packaged up. In designing a large dam an engineer might use soil mechanics to calculate the effects of the water and dam on the surrounding ground. However, if the job were building a two-storey house, soil mechanics would rarely be applied directly. Instead the builder would use packaged principles: perhaps six inches of concrete are needed over clay, but two feet on sand. The builder needs neither an understanding of soil mechanics nor a long experience in trying different foundations to apply the principles. The comparison between the dam and the house foundations is similar to that between a full formal specification and the use of action–effect rules.

Finally, we come to the lift example (§11.2). This is really a sort of "back of the envelope" application of formal models. Formal techniques are applied in a similar style elsewhere. A professional engineer would know from previous formal calculations where stress is likely to be greatest on a beam, and could estimate the likely failure modes of a structure without performing all the intermediate calculations. Some years ago, I used to work on electrically charged fluids. Frequently the question arose: could we ignore say surface tension, or viscosity in a particular circumstance? A quick order-of-magnitude calculation based on an average flow rates, surface radius, etc. was usually

sufficient to rule out one or more effect. It was not necessary to calculate the precise shape of the fluid surface, or the flow characteristics. It is precisely these "back of the envelope" calculations which mark the *professional* application of formal techniques.

So let us finish with this paradox: the more informal the application, the more professional expertise in formal models is required.