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

CHAPTER 12

Conclusions – mathematics and the art of abstraction

12.1 Introduction

In Chapter 1 I said that to formalise is to *abstract*. Although formal statements are often couched in mathematical notation, the heart of the formal statement exists independently of the notation and may often be captured just as well in ordinary English.

Since I first wrote that, several years ago as part of the introduction to my thesis, I have heard several people define formalism as precisely the manipulation of notations independent of meaning. That is, a manipulation of the *form* only. The proposer of such a view then immediately throws away any pretension of dealing with real people or real things. Now that sort of statement immediately sets teeth on edge, but by a dictionary definition I suppose it is correct. In which case I don't really want to be a formalist at all!

In retrospect what I should have written was:

In essence, *mathematics* is to abstract ...

In, short I use formal notation because it is useful, but the mathematics is not restricted to the formalism ...

The trouble is, if I had been writing about *mathematical* models of interactive systems I would probably have frightened off even more readers! However, as you are all now reading the conclusions I can safely nail my true colours to the mast and become a card-carrying mathematician.

The interplay between mathematics and formalism is subtle. Although a mathematical argument may begin with an imaginary inner tube, or something equally non-formal, it is usually assumed that it will eventually be transformed into a rigorous formal notation. When applying mathematics we may either

formalise our problem and attack it using formal mathematical techniques, or alternatively "lift" a formal result into the problem domain and apply it at the informal level. Probably, a mixture of approaches is most common.

Looking at the previous chapter, the first approach, of formalising the problem, corresponds to the specification of a system and relating the abstract models to this specification. The second approach, lifting the formalism, corresponds to the "professional" application of the models as exemplified by the analysis of the lift (*sic*) system.

The major problem we are faced with when dealing with a situation mathematically is precisely what to put into the formal notation. What is the appropriate abstraction? In Chapter 4, we began the formal discussion by focusing on windows as having *content* and *identity*. We then went on to produce a model which encompassed these two features. There was mathematical work needed at both these stages. Choosing the appropriate way to represent content and identity of windows is not easy; there are many equivalent representations, and depending on which we choose the statement of various properties becomes more or less hard. However, I am never too worried about transformations between equivalent mathematical formalisms, and the most important step seems to be the identification of content and identity as central features. That is, a mathematical step of *abstraction*, which is, at that point, entirely in the informal domain.

This example shows us that the job of "bridging the formality gap" requires work on both sides of the divide. We choose to abstract, to look at things from a particular perspective, on the informal side; and on the formal side, we choose an appropriate model or notation which matches the abstraction. There are various heuristics which can be applied to the latter task: identifying entities and operations, etc. However, the mathematical, but informal, task requires more creativity. If abstraction is the essence of mathematics, then developing abstractions is the *art* of mathematics.

12.2 Abstractions we have used

Let us now look again at some of the issues in the previous chapters focusing on abstraction. We will begin by looking at abstraction in the sense used in describing models as abstract. This is, in essence, abstraction by the designer of multiple systems. First (§12.2.1) we will look at how the abstraction chosen can limit the *domain* of our models, that is, the types of statements and systems that the model supports. Then (§12.2.2) we will see how abstractions introduce a *grain* into our analysis. That is, we are led along certain directions by the nature of the abstraction.

Any particular system can be viewed at various levels of abstraction. This is considered in §12.2.3. Both the designer and the user will have such abstract ideas about the system which may or may not coincide. These abstraction are basically intellectual ones, and in a sense dig into the system *away* from the interface. They lose detailed information about the interface and tell us about the inner functionality.

The views we have of a system are, in a sense, the opposite: they are abstractions of the system *towards* the interface. The most abstract such view is that of the concrete pixels of the display, and we move inwards towards less abstract views of the system such as the *monotone closure*. These abstractions of state and display are discussed in §12.2.4.

Finally, in this section we look at the ways abstract models are applied to specific systems (§12.2.5). We see that the most important decision to be made is the level of abstraction at which to apply our models.

12.2.1 Abstract models – domain

Of course, we have laid great emphasis on the abstract nature of the models presented. This was to help achieve generality and to focus on the relevant aspects of the systems. Of course, the generality is never complete and each model has its own *domain* of applicability. We started out with the PIE model. This was claimed to be very general, encompassing nearly all interactive systems. However, as the book developed we have seen many more models addressing areas which were inexpressible within the PIE model.

Some of the restrictions in the domain of applicability were because of the *level* of abstraction. That is, the PIE model was just too abstract to express certain important features. For example, we were unable to address adequately display-mediated interaction and had to look at more architectural models such as dynamic pointers (Chapter 8). Earlier than that, we developed a model of windowing (Chapter 4). The original PIE model was an abstraction of this, as the user's mouse and keyboard commands could be related to the screen without explicitly talking about the windows. However, although the PIE model *could* describe the external view of such systems, a more refined model was required to discuss the more detailed properties of windows as entities in their own right.

Now, this limitation due to level is only to be expected: the purpose of abstraction is to lose some detail and if we later decide we want to talk about that detail we must clearly use a different (more refined) abstraction. The new abstraction may be a direct refinement of the original (as with windowing) or have a more complex relationship (such as dynamic pointers). It may even be that two completely different abstractions are used to address different aspects of the system: for instance, we may use an abstract model of interaction whereas

someone else may be using an abstract model of system security or safety.

Limitations due to the level of abstraction restrict what we can say about a system. The other limitation in domain comes about when the model chosen implicitly restricts the *range* of possible systems. That is, the model is only capable of describing a subset of possible systems. If we again start with the PIE model and then look at the temporal model of Chapter 5, we see that the PIE model was an abstraction of the τ-PIE but only under certain circumstances, when the system admitted a steady-state functionality. Thus not only did the PIE model abstract away from the detailed interleaving and timing of system inputs and outputs, but it *implicitly* limited discussion to systems with a stable steady-state behaviour. For instance, the PIE model is incapable of describing most computer games. We see a similar example in Chapter 10: a general model was given of status input, and then a second model which implicitly required trajectory independence. In fact, it was this second model that I first wrote down, and later, when I realised its limitations, I began to frame the concept of trajectory dependence.

In both cases the formulation of the abstract model limited the range of systems that could be considered. Whereas the limitations due to level reduce *expressiveness*, these limitations of range reduce the *generality* of the models. Now since one of the declared purposes of abstract models was to make generic statements, this loss of generality is rather worrying. The picture is not quite as bad as it seems; there are good reasons for restricting the range of models. Abstract models can be simpler because they abstract away some features of the system; in the same way, models of limited range may be simpler than more comprehensive models. If we are *not interested* in the systems that are not covered by a particular model, then it is more appropriate to use that simpler limited model than a more complicated but complete one.

A more serious problem is that the limitations in range may not be obvious. The level of abstraction may hide the limitations and effort may be wasted on a model that misses the very systems of interest. Further, if the model is used as an architectural framework for developing the specification of a particular system, the system will be limited in scope by the assumptions implicit in the model. Suppose we chose to develop a system using the pipeline architecture (§7.2.3), where input parsing and display generation are seen as separate activities. We saw that the architecture does not support display-mediated interaction (§7.4) and hence the system developed would lack any sense of direct manipulation. More importantly, this central design decision may never have been explicitly made; instead, the structure of the model will have led the design in that direction. This danger of implicit design decisions is a problem not just of abstract modelling; any design system we use will have such implicit assumptions. Application generators will usually produce very stereotyped systems, and interface design methods will only support certain styles of

interaction. Even programming languages will limit the sort of interfaces that can be produced; for example, they may lack real-time programming facilities. The important thing is to be aware of the range of the abstract model (or whatever else) one is using.

Limitations in range can be turned to advantage. If we choose to develop a system based around a model, we do not have to maintain explicitly the properties which are implicit in the model. So if we are designing a mouse-based system and want all commands to be trajectory independent, we can use the model in Chapter 10 which supports only trajectory-independent systems. Similarly, we may enforce appropriate steady-state behaviour by designing the major part of a system using a model (such as the PIE) which does not allow the expression of more complex real-time behaviour. As a separate stage we can map this model into a fully temporal one with due regard for the additional features. This is precisely the design approach suggested at the end of Chapter 5.

It is non-determinism which enables abstract models to describe complete systems whilst ignoring detail. That is, it allows us to increase *expressiveness* without too great an increase in complexity. It is not so obvious that we can increase the *range* of a model by use of non-determinism. However, this is precisely what happened towards the end of Chapter 7. We had a linear model of interaction that was (as we have already noted above) incapable of expressing mediated interaction. However, we were able to increase its range and generality by using *oracles*. So, by adding non-determinism, effectively the same model could describe not only the original range of systems but also a whole range of display-mediated ones.

12.2.2 Abstract models – grain

We saw above that models are restricted in *domain* in terms of both *level* and *range*. We also noted that the most dangerous part of such limitations is when we are not aware they exist. However, there is a far more subtle way that a model may influence the sort of systems that are produced. It may be *possible* to model a large range of systems within a given framework, but the framework naturally steers us towards a certain class.

I pick up a piece of wood to smooth with a plane, look along it, perhaps feel it with my fingers. I try to work it in one direction; the plane sticks, then jumps, taking an unsightly chip off the surface of the wood. So, I turn the wood around and begin to plane in the opposite direction; shavings curve gracefully off the wood and pile as a rich carpet around my feet. I have found the *grain* of the wood. Of course, it is possible to plane the wood against the grain, if you are careful, if the tools are sharp; but it is not easy.

Any material, set of tools, or language will have such a grain. It restricts what can be produced or expressed, not by what *can* be achieved, but simply by what is easy. Perhaps there are exceptions, like carving candle-wax, but they are rare. In particular, of course, the models and concepts presented in this book and formalisms in general induce a grain into the design process.

There is a much greater awareness of the effects of grain when using physical materials than with software. Indeed, part of the aesthetics of many crafts is the way that the intended function of an artifact reflects the inherent dynamics of the material. A Windsor chair produced out of steel would be both inappropriate and possibly structurally weak.

If we question the proponents of computer notations or formalisms there is far less awareness of these issues. We are frequently told that the particular method is all-embracing and general. If the proponent is of a formal nature, this may take the form of a proof of Turing completeness. Alternatively, if we suggest a particular system for which the technique seems ill-suited we get a reply of the form "ah but – you can get the effect by...". The fact that it is *possible* to achieve the desired effect clouds the fact that we *would* not develop such a system using the notation. Only recently, I read a remark that we could do without a range of different programming languages, Cobol, Fortran, Ada etc. because C++ could do it all. I can only assume that the author had never worked at a data-processing site!

How is this reflected in the models in this book? In Chapter 10, I said that it was possible to describe status input behaviour using the PIE or τ-PIE models. These models did not, however, immediately suggest status input, and we would not tend to design systems with status input if we had these models in mind. So they have a definite *grain* to them.

Where does this grain come from? Well, it is partly in the formal expression: by not separating out status and event inputs in the model, the possibility is not suggested by the formalism. Also, aspects of the formulation such as the use of sequences suggest discrete events, and the types of principles stated tend to be appropriate for events rather than status inputs. Moreover, the examples used were of event input systems which tended to limit the way we thought of the system.

In Chapter 8, we can see another example of the way examples influence the way we think of a model. All the earliest examples of dynamic pointers were of atomic pointers. Only later in the chapter did we get onto block pointers. Now partly because of this, and partly because of the connotations of the word *point*er, I have found that readers assume that block pointers are somehow different, rather then just a special case of dynamic pointers. The formal expression of pointer spaces with *pull* maps and projections with their *fwd* and *back* maps apply equally well to the atomic and block cases, but the examples and the nomenclature bias the reader.

If we were looking for a piece of wood from which to make the prow of a ship, we could just take any piece and steam-bend it. However, steam-bending tree trunks is rather hard work. Alternatively, we might look for a piece of wood (or a tree) that already had the desired shape. Of course, it wouldn't be exactly right, but we would be working with the grain of the material, not against it.

We can do the same with interactive systems. We saw that we can deliberately choose models which constrain a design to have certain features. In a similar way, we can choose a model or notation so that its grain encourages a style of design. This happens in various contexts: an interface toolbox may in principle allow the designer total freedom but actually encourages a particular style of interaction. Many programming languages are largely sequential in character, and I have shown elsewhere that this leads to systems where the computer has excessive control over the dialogue. As a deliberate attempt to counter this, I showed how a non-sequential computer language could be augmented with input/output primitives in a non-standard way deliberately to encourage user-controlled dialogue.

The red-PIE model was always intended to be applied at many different levels of abstraction, and to various parts of the system. Now although this message largely got across at the level of overall abstraction, so that people felt comfortable with the display being a bit-map or simply an abstract description, it has never really been successful when applied to facets of a system. Other workers at York were interested in the way that certain parts of a screen and of the result were salient to the user at different times (Harrison *et al*. 1989). They emphasised this by having functions called *display templates* and *result templates*, which abstracted from the current display and result the features that were important for a particular task. Now arguably these are unnecessary; we could just apply the red-PIE model choosing the display and result to be the appropriate bits, and relate it back to the complete system by way of an abstraction relation. However, although the templates may not be strictly necessary in terms of expressiveness, they make explicit a feature of the system and therefore bring it to attention. That is, they have created a model with a certain *grain* which encourages systems where the subdisplays and subresults corresponding to tasks are well thought-out and discriminated.

Now this issue of grain is interwoven with the concept of *structural correlation* that I have talked about previously. One of the most obvious forms of grain is where the structure of the model influences the structure of the system. Indeed, the paper I cite above, where a language was designed to encourage user-controlled dialogue, was written in precisely these terms. The critical point was the inherent structural correlation between the programs and the ensuing dialogues, and hence between the programming notations and the interface. So, when we cross the formality gap between what we feel we want and formally expressing it, we must be careful that the grain of the formalism

flows with us as we produce our description.

12.2.3 Abstractions of the system

The issues we have discussed, the expressiveness and generality of a model, are about the abstraction of the model. However, we can look at a particular system at various levels of abstraction. The models may be appropriate at various levels and may even help us to describe the levels of abstraction within a system. This is essentially what was going on in Chapter 7. We wanted to view a system at the physical level and at an inner, logical, application level. These layers of abstraction were actually represented by a pair of PIEs with functions between them. That is, we used an abstract model to talk about abstraction!

It is common in both HCI and software engineering to discuss systems at various levels. In the latter, the levels correspond to layers of the software system that isolate the true heart of the system from the physical input and output. In the former, the levels are more interesting as they may correspond to the way the user construes structure from the interaction. I say *may* as they may equally well be a re-expression of the system side; however, to be generous we should be thinking of levels of abstraction *for the user*. On the input side (computer's input, that is), this ranges from some level of tasks or goals through various levels to the actual movements of the user's fingers on a keyboard. On the output side, this includes the actual seeing of a screen through to the recognition of the content of the display as pertinent to the user's intentions.

The user is often unaware of the lower levels of abstraction of the interaction. Much of the time I am "writing a book", only partly aware that I am also "using a word processor". I am not thinking that I am "typing at keys" or "reading characters", and certainly not "moving muscles" or "interpreting light patterns". In fact, I am probably incapable of even performing these last actions at a conscious level. The times I become aware of the lower levels of abstraction are regarded as "breakdown", often an exposure of faults in the system.

There is a general hope that levels of abstraction for the user will correspond reasonably well to the layers within the system. My understanding of the system matches what is there. Now in §11.5 we saw that this was in general impossible. The pragmatics of producing reasonably efficient systems means that the systems must match structurally the demands of the machine. If the layers understood by user and system are to agree, then the user must adopt an implementationally efficient view of the system (or buy a supercomputer). Sadly, this is the state of much software: the user is taught to interact as a machine.

Happily, the analysis of refinement led us out of this impasse. The *specification* of the system can be layered in a way which presents a natural structure for the user, whilst the *implementation* of the system may adopt a different structure. The techniques for managing this structural change are

especially important given that the implementation layers refer to the way the system is built. In Chapter 7 we saw that the abstraction of a system which we could regard as the functionality is in general *not* a component of the system as a whole, but an *abstraction* of the system. By appropriate transformation we can retain this abstraction relationship within the early specification and move to the constructional relationship as development proceeds.

Movement between levels may occur not just because of breakdowns: we intersperse high-level, more abstract, planning with more concrete, less abstract actions. It is frequently the case that these movements in level are associated with syntactic units in the grammar of interaction.

Examples abound: in my particular case I decided to edit a particular section of this book, so I invoked the editor; when I have finished I will exit it. These shifts in level between the abstract "edit the section" and the more concrete actual writing are reflected in the interaction, and could easily be found by anyone with a trace of the dialogue. Within my editing various subtasks occur, perhaps a global replacement of a habitual misspelling. Again the shift in level is reflected in the dialogue.

Now the levels of abstraction by which we understand a system need not agree with the levels by which it is implemented (nor even necessarily specified), so long as the models are consistent. Of course, the place that this consistency must be found is in the physical interaction itself. So the system need not interpret the structure of our interaction in the same way as I do, but it must at least be consistent with my interpretation. (Or to be precise, I suppose I must be consistent with it!)

Assuming principles of structural correlation hold and the designer's model of the system agrees with the user's, we can relate the movements in the user's level of abstraction to the layering in Chapter 7. The points in the dialogue when the *parse* function yields abstract commands correspond to the shifts in level. Other workers in York have formalised this in the concept of a *cycle* (Harrison *et al.* 1989). The whole interaction is looked at as a sequence of cycles. Each cycle affects the result of the interaction, but no major changes occur within the cycle. At one level of abstraction, each cycle can be thought of as one abstract command. This corresponds to the inner level in §7.2.1. The cycles can have subcycles which themselves can have subcycles. This exactly parallels the stacking of layers of abstraction within a system. They are particularly interested in the ways by which the user can tell when a cycle has finished: that is, how the user and the system *synchronise* their level of abstraction. They focus on the clues in the display which are repeated when cycles are complete, such as main menus or prompts. At this point they can begin to make prescriptive statements about the necessity of such clues to interaction.

To summarise: users and designers must agree statically on the abstractions by which they understand the system, but they must also agree dynamically as the interaction proceeds as to which level is current.

12.2.4 Abstractions of the state and display

In several places, particularly in Chapters 3 and 9, we have been interested primarily in the output side of the system. Arguably, Chapter 9 on views was about input as well: as we were interested in the way we could update the state through the view. However, we were concerned not with the dialogue, just the relationship between the internal state and views of that state. The treatment of different forms of predictability and observability in Chapter 3 was of a similar form. There were issues of dialogue hidden within the discussion, in particular the *strategy* used by the user to view the system. However, these were effectively hidden by packaging up this dialogue in the *observable effect*, a static view of the system's state.

In both places the prevailing ethos was of the output as an *abstraction* of the state. In user-oriented terms, we could say that the user is aware of an abstraction of the state of the system at any time, and these models capture this awareness in different ways.

Note that the "state" that the red-PIE and the views model capture are very different. In the red-PIE model the state was the *entire* internal state of the system. Now it is a truism that the display and results of a system are abstractions of the internal state, as the state of the system includes all the memory locations that hold the screen display, etc. The red-PIE discussions therefore focused on the relationship between various system outputs.

In the views model the basic state we dealt with was very different. We called it a "database", partly because of the interrelationship with database theory, but partly because this was not the entire internal state but merely the state of the "objects of interest". The additional state due to the dialogue, physical display, etc., was ignored. Note that the very fact we were talking about the user seeing *views* of this state was prescriptive, and is an example of a implicit restriction of the *range* of the model.

State and monotone closure

Of course, we may look at many abstractions of the internal state of the system. For instance, the programmer will have access to some of that state, but may not know about some aspects such as the precise screen bit-map, or partly buffered user input. From the user's point of view the *monotone closure* introduced in Chapter 2 is particularly important. For any viewpoint, it gives precisely the most abstract state of the system which is still capable of telling us everything about the future behaviour from that viewpoint. I know that this

concept has caused difficulty with many people, but it is incredibly useful in expressing and understanding interface behaviour. So we shall spend a short time with monotone closure and abstractions of the state before discussing displays and views in more detail.

Part of the difficulty of the monotone closure is that it does not belong to any one or any part of the system. The display and result of the system are things that the user can find out about. In a sense the user *has* these abstractions. Similarly, the levels of abstraction in the system *belong* to the designer, in that they are explicit parts of the specification. The system itself may also directly express levels of abstraction, even if these are different from those expressed in the specification. The monotone closure is different. In general, it is uncomputable. If this is the case, then it is impossible to look just at a bit of the system and get it as abstraction. It is there, but inaccessible. The designer may be able to talk about it, but may not be able to say what is in it. Similarly, it affects the behaviour of the system as seen from the user's viewpoint but may not be directly viewable from that viewpoint. It expresses all the *potentiality* of the viewpoint, but no amount of exploration need uncover all that potentiality. Remember, this is not philosophising, just the outcome of a formal procedure, but it does perhaps have a message that we can apply to other domains.

Now although mystery is a normal and important part of everday life, it is an element that we usually try to minimise in interfaces. So, *in general* it may not be possible for a designer to elaborate the monotone closure for a particular view of a system; however, we would expect as a normative requirement that most systems would be susceptible of such elaboration. That is, we require as a part of the design process that the monotone closure of each view of the system is given. At a formal level this sounds a little heavy, but it really comes down to asking the fairly basic question: "what parts of the state can possibly affect this viewpoint?". The question is not as easy to answer as it seems because we must consider all possible future inputs to the system. However, if the designer does not know the answer to the above question, there is probably some deficiency in the design.

We would relax this requirement a little for the inner functionality of the application itself. For instance, we might have a system that took as inputs triples of numbers, (x,y,z). At the nth input it would print a one or a zero depending on whether:

$$x^n + y^n = z^n$$

Now assuming Fermat's Last Theorem (Hardy and Wright 1954) is true, the monotone closure of the state after the second input is void: all future outputs are zero, thus no state is required. If, however, Fermat's Last Theorem is false, we need to keep a count of the number of inputs so we know the value of n to compute the formula.

So, we would look for computable monotone closures for the *interaction* state of the system but not necessarily for the *application* state. Of course, this distinction is particularly hard to maintain; (cf) to be totally formal we would have to talk about a non-deterministic system state where the non-determinism corresponds to the results of the application. In the example above, the truth of the expression would be regarded as non-deterministic. This, of course, requires an extension of the definition of monotone closure to the non-deterministic systems as described in Chapter 6. Such a definition was not given; it is fairly easy to produce a generalisation for the non-deterministic case, but a little care is needed. However, I would imagine that total rigour is unnecessary; the formal analysis has told us what questions to ask, and the designer will have enough wit to decide which elements of the state are part of the application and unknowable, and which are the domain of the interaction.

Display and result

We can move on to the display and result as abstractions of the state. Because they were abstractions of the state, they had corresponding monotone closures $D^\dagger$ and $R^\dagger$ (§3.2.3). The result's closure, $R^\dagger$, constitutes *all* we need know about the state in order to predict the future results of the system. In particular, if we had an operating system with a "browse" facility for viewing files, while we were using the browser its internal state would be completely ignored as part of $R^\dagger$. No changes could be made to the result of the system (presumably the file system) and hence it would not be part of $R^\dagger$. The only thing we would require would be sufficient state to mimic the exiting of the browser.

From a task point of view, $R^\dagger$ could be seen as *the* important abstraction of the state, as it predicts exactly the future behaviour of the result. However, the result may be the endpoint of the task, but does not constitute the whole *interactive* task. The monotone closure captures only how the system will behave *if* the user enters any particular inputs. Of course, the behaviour of the interaction depends on which inputs the user enters, and hence on the other outputs the user receives. This same point was made when we considered window independence in Chapter 4. This is clear from the browser example: the future behaviour of the system may not depend on what the user does with the browser, but the future behaviour of the user will.

The display's closure, $D^\dagger$, is all we need to know about the system in order to predict future displays. We are on much safer ground in asserting that this is precisely the important abstraction of the state for considering the interaction. The behaviour of the user and computer together is completely determined (from the computer's side) by this. Elements that may be in it, but not in $R^\dagger$, include the state associated with browsing, as above, elements such as clocks or pop-up calculators, and less desirable features like mistakes in display update.

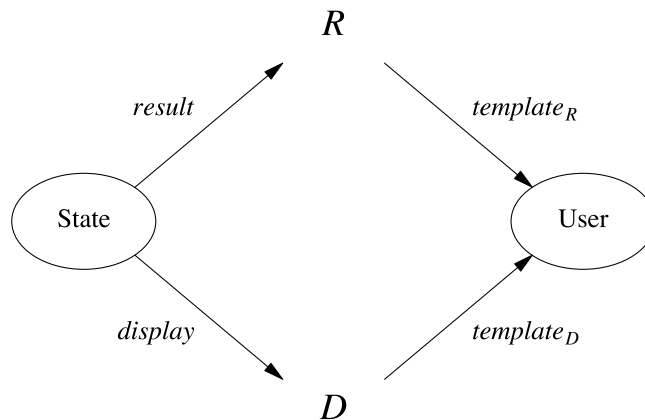
The closure $D^{\dagger}$ is what affects the display, but may in general be inaccessible to the user. The *observable effect* (§3.3) was precisely the abstraction that the user was able to obtain from the system. It is an output abstraction of the state: it is not an abstraction of the actual outputs at any point in time, but, like the monotone closure, expresses a form of potentiality. Unlike the closure, which expresses the potential effects of the system on the user, the observable effect expresses the user's potential for finding out about the system. It is thus an encapsulation of the user's control over the disclosure of internal details. It emphasises the user as an active participant rather than as a passive recipient.

It is precisely because the potential expressed in the observable effect captures the user's control, that we sought to have principles that showed that those features that might affect the user were observable from it. Predictability and observability properties assert the user's mastery over the system.

In practical design we would not want to take a system and then try to prove properties about it. Instead, we would look at the system and seek to make it satisfy the predictability properties. We have already thought about how the designer can ask about which parts of the system may effect the user. Having done this, the next stage is to ask: "how can the user know about this component of the state?". The display and command repertoire can then be designed (or altered if this is *post hoc*) to allow the user access to the relevant parts of the state. Of course, in designing this display and dialogue more components of system state will have been added, requiring further analysis...

The above discussion talks about *the* display and result. In fact, the display and result depend on the level of detail at which the system is viewed. For instance, at the operating system level, the result might be the file system and the display an entire workstation bit-map. If we consider an editor used within that operating system, the result is a single file and the display one window within the screen. At an even greater level of detail, we may look at the specification of search/replace strings. The result is the strings themselves, and the display perhaps a dialogue box within the editor window.

The red-PIE model can be applied individually to these cases; however, as mentioned above (§12.2.2), other workers at York have found it useful to apply additional abstraction functions to the display and result. The display D and result R are then chosen to be maximal for the range of tasks of interest, and then for particular subtasks different functions $template_D$ and $template_R$ are chosen to capture what the user is focused on in both domains:



Note the similarity to the perception functions δ and ρ in the WYSIWYG model in §1.5. It is appropriate to add these additional abstraction functions because they make *explicit* within the model the additional levels of abstraction from these domains.

Views

We noted that the state that we considered in Chapter 9 was already an abstraction of the full internal state and represented the objects of interest within the system. In particular, we ignored the interaction state. In fact, the process of producing complementary views and update strategies was part of the process of constructing such an interaction. Thus this process sits logically before the analysis of display and state, although some of the insights have been applied to a PIE-like model by Michael Harrison and myself (Harrison and Dix 1990).

The views are simple functional abstractions of the underlying database, but the sorts of properties that arise give some insight into more complex forms of abstraction. For example, the search for a complementary view is a particular example of the general problem of framing: knowing the unchanging context against which we are to set the additional information we are given. Framing problems occur in many different contexts and functional views are surely one of the simplest.

When we move from the views themselves to the problems of update and complementarity between updates to the view and database, we are extending our abstractions to be abstractions of the dynamics of the system. This is somewhat like the situation in Chapter 7 of layered systems, except there the abstraction was away from the physical system as the user experiences it inwards

towards the system. The abstraction of the system given by views is one where the abstraction is outwards, from the system towards the user. The abstraction is capturing the fact that the user does not see all of the system.

The contrast between views in general and product databases as a special case is a second reminder of the distinction between components and abstractions. There is often a tendency for users of formalisms to miss this distinction and make assumptions based on a component-wise view, when the practical reality is a far more rich abstraction. The change in perspective between a product database with constraints and a totally view-based approach is not simple. They are equivalent, but have a very different feel. When constraints are few, the product formulation is often easiest, and even where views of interest do not fall out as simple components, they are likely to be expressed as functions of components. The major "extra" given by product databases is that the complement to choose for a particular component is obvious. There may be many complements that we can choose, but the fact that a Cartesian product formulation is used *implicitly* encourages us to assume that unnamed components are unchanged. This is a form of *meta-framing* principle, and, thinking back to the previous discussion (§12.2.2), represents a *grain* to the product database model.

12.2.5 Using abstract models

If we want to apply existing abstract models to a particular system, how do we proceed? There are several decisions to be made: which models to use, which principles should hold for this system. The choice of model depends partly on the sort of system we have and partly on the sort of things we want to say about it. So, if we have a windowed system we would instantly think of the model from Chapter 4, but if we are interested only in the temporal behaviour of the system we may not need to talk about the windows explicitly and may be able to use a model such as that in Chapter 5 directly. Of course, we may want to know about both. We could use a model encompassing both windowing and temporal aspects, but most likely we would apply each model to a different *abstraction* of the entire system. When we apply the windowing model we would be considering the steady-state behaviour (assuming it exists!), ignoring detailed timing, and when we applied the temporal model we would consider the relation between steady-state and full temporal behaviour ignoring windows.

So, in general we proceed by associating a model with an abstraction of the system in which we are interested. Each element of the model must be associated with an element of the abstracted system. For instance, if we consider the PIE model and a syntax tree editor, we may want to look at an abstraction that ignores the bit-map display but instead regards the command set C as including operations like "delete subtree", and the display D to be the raw syntax tree. Alternatively, we may want to look at the physical level and talk about the

command set C as mouse clicks and keystrokes, the display being lighted and unlit pixels.

Typically, any model may be applied in several different ways to the same system. Part of the purpose of the discussion of layered systems in Chapter 7 was to capture these simultaneous abstractions *within* the modelling process. However, the *choice* of abstraction and the *mapping* between model and system are not in themselves formalised. Further, for each application of the models we may want different principles to apply. These design choices represent the formality gap in reverse, moving from an abstract formal statement back to the real world. The movement over this gap is not, and cannot be, part of the formal process itself, but is a step that must be accomplished for any formalism to be useful. Throughout this book, with each model and especially in Chapter 11, are examples of applying models to various situations. There are heuristics and guidelines, but the final step is always one of creativity.

For an example of this in another domain, we look at simple mechanics. In principle, given the moduli of elasticity of a material it is possible to calculate its deformity under various loads. One standard problem is to predict the displacement of a bar when a small load is applied to its end. I remember in school struggling for a long time with this calculation, but in vain, as the equations I generated had no easy solution. My problem lay in the approximations I had made. For small peturbations there are many different equivalent ways of describing the system, and only by selecting the correct one does the solution fall out easily. That is, the solution is obtained by applying the formal model of elasticity to the right abstraction of the system.

As I said at the introduction, developing general abstractions is an art, perhaps the art of mathematics, and it is by this means we develop formal understanding. The corresponding art of applying mathematics in general, and abstract models in particular, is choosing the correct abstraction to which to apply this formal understanding.

12.3 About abstraction

Abstractions can be tricky things. The further we get from the concrete, the easier it is to make silly mistakes, but there again the more powerful the things we can say. We have seen how abstraction is a common theme running through this book, and so in this (nearly) final section, we shall look at some of the misunderstandings and dangers of abstraction and formalism.

As well as being central to mathematics, abstraction is central to all language, both natural and esoteric. We can only communicate by abstractions, ignoring what is irrelevant in order to talk about what is relevant.

So, although abstraction and formalism have problems, they are often problems shared by more "informal" approaches. On the other hand, abstractions, both in language and in formalisms, have the potential of richness and power.

We begin by looking at the belief that mathematics and formalisms are precise and therefore form a useful means of communication. Both beliefs are seen to be false: formalism is by its nature ambiguous and, unless that ambiguity is addressed, may block communication.

We then go on to look at problems due to an uncritical acceptance of abstractions and formalisms. Both are powerful and useful in developing our understanding of the world, but must be used with due consideration for their limitations.

Finally, we look at relationships and analogies. We see that some of the richest uses of formalism are when the formal model is seen as an analogy of the real world, rather than drawing out the precise formal relationship.

. . . . .

Before moving on here is a short story: its relevance will become clear further on. Read it through once and see if you can answer the question at the end. The correct answer is given later, but be careful as you read – don't be distracted by irrelevant details.

Tom Tipper the tipper truck driver went out one day, delivering sand. In the back of his tipper truck were six piles of sand. He stopped first at Miss Jones' house, where she was building a barn to house her collection of vintage leeks. She had ordered three piles of sand, but decided she really needed an extra one as well. Tom shovelled out four piles of sand and went on to old Mr Cobble who wanted one pile of sand to make concrete gnomes with. After that Tom called back at the yard to pick up another four piles of sand as his tipper truck was looking a bit empty.

Tom Tipper had a quick sandwich and drove out along the bumpy track to Farmer Field who was making a new patio for her husband. Tom shovelled out three piles of sand for her and then went for his final call of the day. "I really wanted five piles of sand for my new cell," said Mr Plod, "but I'll just take what you've got." So Tom tipped out the rest of the sand and went home for his kippers and ice cream.

How many piles of sand did Mr Plod have to build his new prison cell with? Remember your answer for later.

12.3.1 Ambiguity and precision – the myth of formalism

If you asked a software engineer who was "into" formal methods to list their advantages, the list would probably include a statement like: formal methods provide a precise means of describing systems and communicating to others. Although there is some truth in this, it is deceptive, and potentially dangerous – *formal descriptions are inherently ambiguous*.

Some readers may have already thought through these issues and may find this fact self-evident; I think many will not. To the convinced formalist it may seem almost heretical, and to others plainly false: surely the whole nature of mathematics is precision. It is one reason why many with a background in the "soft" sciences or the humanities dislike formalism: there is too much precision to allow for the complexities of real life.

However, mathematics *is* ambiguous precisely because it is founded on *abstraction*. The nature of abstraction is to ignore detail, to focus on one aspect of the world at the expense of others. So it trades total precision about some aspects at the cost of utter ambiguity of others. This is no bad thing – it is the power and strength of mathematics – but it is a wise thing to bear in mind when you use it.

Consider the simplest form of mathematics. You have two oranges and get one more orange. You now have three oranges. You have two apples and get one more apple. You now have three apples. Abstract:

$$2 + 1 = 3$$

We now know something general that can be applied to all sorts of things: oranges, apples, atoms of hydrogen, sheep, greater-spotted aardvarks, bank balances, piles of sand... oh yes, piles of sand.

If you have forgotten your answer to Tom Tipper's quiz, you could have a quick look back now.

How many piles of sand did Mr. Plod get? Two? The correct answer is one. Tom Tipper tipped out the remaining sand. It fell, of course, in one (big) pile.

It's Christmas day, there are two plates, on one are two roast parsnips and on the other, three. Tom Tipper likes roast parsnips a lot, which plate do you give him? I hope you are prepared now, it depends how big the parsnips are.

The laws of arithmetic depend on the items being regarded as "units" which retain their identity and are each to some degree equivalent. But two piles of sand may become one pile of sand, and one roast parsnip may be bigger than another. All oranges are created equal...

It is not just the mathematics which causes problems, even terms like "roast parsnip" and "orange" are themselves abstractions. The use of numbers helped to steer us away from the reality of the precise parsnips involved, but as soon as we said "parsnip" the actual golden steaming roots on the plate were lost. To be utterly precise we must be rooted in the individual items and moments of existence, but that rules out all communication and understanding.

So if we want to communicate we need to use words like "orange". The word "orange" is ambiguous. There are many different oranges, of different sizes, shapes and tastes. But when I say "orange" it means something to you. In a book I cannot gesture at my fruit bowl. My understanding of, and ability to discuss, oranges in general inevitably carry the possibility of misunderstanding about precise oranges. Abstraction and ambiguity go hand in hand.

To go from parsnips to programs, we are bound to lose something as soon as we move from the precise system to talking about systems in general. Indeed, even to talk about the system without regard to a single use in an explicit environment is an abstraction. Some workers in HCI would take precisely this view, that interactive systems cannot be discussed in general but must be experienced in context. In the light of the above discussion, this is undoubtedly true, but I'll bet they add up their change in the supermarket.

The level to which we abstract depends partly on the situation, partly on our individual temperament. One thing that we must be aware of is that we always abstract, and that we are always ambiguous. What is essential is to know that the ambiguity is there, to know just what is being abstracted away, and what the limits to that abstraction are.

Neither formalist nor non-formalist can be smug on this point. We all abstract and we all forget at times that we have done so. The special danger for formalists and for those listening to them is when either side believes the bald statement "mathematics is precise".

This brings us back to communication. We need abstractions to communicate, but forgetting about the inherent ambiguity can destroy that communication entirely. It has frequently been the case, even when working relatively closely with colleagues, that a new model has caused considerable trouble. Take the

windowing model from Chapter 4. When I first showed this to a colleague I wrote down the formal model, the sets and mappings, and then started to discuss formulation of principles. As the discussion proceeded there was obviously a growing level of misunderstanding. We agreed about the formal model, it was written there before us. But we differed on the *interpretation* of that model. To me, the "handles" in the model were merely place-holders to denote the identity of the windows. My colleague was interpreting them as functions which extracted the contents of the windows (like the views in Chapter 4). The difference was not important for the static properties of the windows, but (as we saw at the end of Chapter 4) it became so when we wanted to consider dynamic properties.

Now if we were engaged in Pure Mathematics this might not have mattered. We could both have seen the correctness (or agreed about the errors) of the formulation. Internal consistency would have been sufficient. (In fact, this is rather a caricature, even of Pure Mathematics.) However, as soon as we start to say things like "this would be a good principle" or "an obvious extension of the model would be...", things become more complicated. What makes a good principle depends on meaning. External consistency is paramount. The formal model captured denotation but lacked connotation. Although there are limits to shared understanding, we must have some level of common meaning to the words we use.

Now if we were talking in everyday English, we would perhaps have been more prepared for problems of language. It is well known that many arguments boil down to a different interpretation of words. The problem with the formal description was not that the ambiguity occurred, but that the myth of formal precision made us unprepared for such misunderstandings.

On the other hand, common knowledge of the ambiguity of language does not seem to stop it being a problem in many situations. One advantage that formal models seem to have is that the ambiguity, *once we realise it is there,* is more easy to pin down.

The problem my colleague and I faced could also be described as *meta-aliasing*. Simple aliasing happens when two things (windows, views, parts of documents) look the same but are different: or, in the language of Chapter 4, content does not determine identity. Here we have a similar problem, but at the level of talking about systems. Two abstractions of the system have the same formal expression, but are different. In common with other types of aliasing, the similarity will break down sooner or later.

So formalisms capture some properties of interest, but divorce themselves so much from the things they represent that ambiguity arises, not just ambiguity about what is ignored, but even ambiguity about which abstraction is being used. This is very similar to the sort of problems we had with views in Chapter 9:

Aliasing

which view – which abstraction

Complementary views

what don't we see – what is abstracted away

Views deal with abstractions at the concrete level of what the user can see of a system. Formal models are abstractions at the level of talking about systems. Both share many properties and, having seen the problems at the concrete level first, we have been perhaps a bit better prepared for distinguishing and discussing them here.

If formal notations can lead so easily into misunderstanding, why do some people claim to find them such useful tools for communication? My guess is that in all the situations where this is claimed the notation is shared by a closely knit design team. The meanings of each part are known by all, as the formal language is built in a community. Similarly at York, although there may have been problems as described above, once a common understanding has been gained the models form a rich common language. However, I deeply doubt the scenario painted by some software engineers, of a formal analyst producing formal specifications which are signed over to some lowly programmer to implement.

Unfortunately, you, the reader, were not part of this group wherein the common understanding of these models has grown. How have you managed to get this far in a book littered with formal models? Of course, if I wanted to learn Urdu I would not just grab a few books written in Urdu and stare at them hopefully. I would want either to see Urdu juxtaposed with English, or to hear it spoken in context. Just so when we deal with formal notations: the model or specification must be developed in context; meaning must be attached to the formal symbols.

Although many formal notations pay lip service to well-documented specifications, as far as I am aware only Z has included this as part of the standard language. A Z specification is always part of a document which mirrors in natural language the progress of the formal specification. So formal notation and meaning are acquired in parallel.

With specifications of concrete systems, the natural language descriptions are concerned chiefly with explaining complex formulae. The basic symbols often inherit their names from the real systems that are described. So if we have a specification that talks about a *screen* being an array of 80×25 characters and a *cursor* being a pair of integers, we will probably know what is meant by these symbols with little further explanation.

The more abstract the formal description is, the less easy it is to attach meaning to the symbols. Even if we know what is referred to by a term such as "window", we do not know what level and way it is abstracted. Any sort of communication of formal models must therefore be full of examples and textual or verbal description.

Looking back over this section, it all seems almost too obvious to bother writing. However, the myth of formal precision persists and so some reminder is obviously necessary.

12.3.2 Uncritical dependence on abstraction

We have already seen how some problems can arise if we forget about the ambiguity inherent in abstraction, but there are other problems associated with abstraction and with formalisation. Many of the problems can be recognised both in the application of formal methods to interaction and in everyday life. Perhaps recognising problems in the relatively simple world of formal models may help us to understand more complex issues.

One of the reasons for introducing abstract models was to *generalise*, to state properties of whole classes of interactive systems (§1.2). This means we can recognise a property in a particular situation, see how this is represented in the abstract model, and then apply this generalised property to all the systems described by the model.

The obvious potential danger is, of course, *overgeneralisation*. For example, many problems in older text editors are due to modes: the same keystroke has a different meaning dependent on the current editor mode. From this we might conclude that modiness is a bad thing, and frame principles of mode freedom in the abstract model. Menu and mouse-based systems are incredibly mode-ridden: virtually everything we do depends on what is being displayed. If we blindly applied principles of mode freedom, our mouse-based systems would become very boring indeed.

Why then did the generalisation fail? Well, even in the process of describing modes above, I began to abstract. I talked about the problem as one of different meanings at different times. The problem is more rooted in the particular situation than that. Modes are a problem, partly because the user may not know the current mode, partly because of remembering different meanings. In the menu or mouse-based system, the user's attention is on a screen which indicates clearly (sometimes) what the meaning of a mouse click or a particular key will be. The presence of visual cues removes ambiguity and prompts recall. In retrospect, we should have talked about the need for visually distinct modes, the user's focus of attention, etc. By framing the problem in terms of meaning of keystrokes, the range of appropriate generalisation is limited to those systems for which these other factors are similar.

Now I hope it has always been clear that the various properties and principles framed throughout this book are putative properties that we may want to demand of a particular system. By the nature of abstraction, these principles ignore factors which are abstracted away, When the principles are applied, these additional facets need to be brought into account.

Note also that the *grain* of the abstraction we are using will tend to make us view a problem or property of a system in a particular way. We are driven by the abstraction to look at certain classes of problems. Choosing an abstraction colours irrevocably what we can say and think. This is at least part of the problem of prejudice: as soon as we decide to use colour, sex or race as the principal attributes by which we describe people, we inevitably use them to discriminate. Alternative abstractions, such as friendliness or generosity, generate completely different world views. Of course, any abstraction tends to hide the individual.

Another related problem is "nothing but"-ism: we use an abstraction to make sense of the world, but then forget that it is an abstraction. Part of my reticence in including the user explicitly in models stems from this. We could describe the user as a non-deterministic function from displays to keystrokes. This would make clear the feedback nature of an interactive system, whilst making no assumptions about the user's behaviour. It does not worry me too much if someone should mistake the PIE model as saying that an interactive system is *nothing but* a function from inputs to outputs. However, I am loath to allow even the suggestion that a user is *nothing but* a non-deterministic function from displays to keystrokes. Nothing but-ism abounds in everyday life. The sun is nothing but a ball of hydrogen and helium plasma – but it is also a God-given sustainer of life. A factory's future is determined solely (or on nothing but) economic grounds – but it is also a sense of community, the livelihoods of workers. Abstractions encourage nothing but-ism, by focusing on certain aspects and ignoring others. When we use abstractions, whether formal or informal, we must make a compensating effort to look at the world beyond the abstraction.

12.3.3 Uncritical dependence on formalism

When I print rough drafts of chapters for this book, they are always typeset on a laser printer. The quality of the earliest copies will be very similar to the finished book. It is amazing how strong an aura of authority well-printed text has, no matter how bad or good the content. The same text printed on a dot-matrix printer (or in my hand-writing!) would carry far less weight. Now formal notations can give a similar appearance of weight and credibility whilst not necessarily adding anything in terms of content.

It is all too common (more so in software engineering than HCI) to see papers where a "formal bit" is included which adds little to the content and amounts to using the notation of sets and functions to carry the same meaning as a box diagram. If the authors had used the equivalent box diagram the text would have been clearer and just as rigorous. Of course, the presence of the equations (especially the odd λ and special symbols) makes the paper look very formal and impressive. The authors probably worked hard to get the correct formal syntax, but if you look closer you find deeper problems. A common mistake is that the functions are not really functions: vital parameters are left out. With a box diagram, this sort of imprecision is accepted, and the use of such a diagram would have been *more* rigorous and accurate. There is clearly no intention to fool the reader, just a half-hearted attempt to use formalism. At the best this merely wastes a little of the author's time and effort, but at the worst can hide the fact that the author has not really developed an understanding of the domain.

Not only can formal notations be convincing in print, an incorrect formal argument may be believed, *however much it conflicts with reality.* The belief that, once something is in mathematics it is correct, is common even in academia.

Some years ago I used to work on the mathematical modelling of electrostatically charged sprays. The basic equations were easy enough to write down, but under most conditions the only way to solve them was by computer simulation. In such cases it is always useful to have analytic solutions to simple cases, as these give one a far better "feel" for the general behaviour, especially when there are lots of different variables affecting the solution.

One obvious special case is to look at spray eminating from the centre of a circle or sphere, with the circumference held at some constant voltage. This would be a good approximation to the centre of a cone of spray. One day, I came upon some papers written by two Japanese *physicists* who gave solutions for precisely these cases. I was interested; I had got stuck each time I had tried to solve these particular situations, and they had even used more complex forms for air drag on the spray. I tried to work out how they had got their solution, but could not reconstruct it. This bugged me for some time, then one day I looked at the sort of results they were obtaining. Some of the outputs from their equations seemed wrong. If you had designed the equivalent physical experiment they would have been *inputs*. So the *mathematical* results did not correspond to *physical* reality. Having noted that something was wrong I went back over the calculations, looking for an error. Sure enough, you could obtain their results if you made a standard school-book mistake.

Now, the mistakes were not complex: I am sure the authors were quite capable of understanding that level of mathematics. Why then did these *physicists* not notice that their results were unphysical, whereas I, a mathematician, could see this? (remember I did not spot the mistake from the mathematics alone). The

answer is that once they had set out their problem mathematically they trusted the mathematics totally and threw away their physical understanding of the system. Even if they had noticed some problem with the results, they would not have dreamt of doubting the mathematics.

Now computers generate the same sort of awe in many people. The combination of formal methods in computing can leave many simply dazed. On the other hand, the opposite attitude of complete distrust of computers and formalism (or of formalism by computer scientists) is equally unhelpful. If formalisms are to be useful one must develop a respectful, but not irrational, distrust of the results. If you used a calculator to add up a supermarket bill and got an answer of £2,073.50, you would guess that something was wrong. Formal models must similarly be constantly checked against one's intuitions about the system. If they differ, both the formal reasoning and intuition can be reassessed. I frequently make mistakes in both realms.

12.3.4 Abstraction and analogy – mathematics and poetry

We have been concentrating on abstraction, both as a unifying theme for the models in this book, and because of its central role in mathematics. However, in any mathematical study there is a stage before full formal abstraction which gives a pointer towards the meaningful application of formalisms.

Earlier, we went through the first stages of developing addition: two oranges and another orange, two apples and another apple... Then apparently the magic step: 2 + 1 = 3. How did this abstraction arise? We saw there was a similarity between the situations we encountered in concrete. The two oranges were similar in a way to the two apples, the extra orange was similar to the extra apple, the resulting piles of fruit were similar. Some things were different in the two scenarios: one talked about oranges, the other talked about apples. The appropriate abstraction is therefore one which abstracts away the points of difference and retains the similarity of structure. Our earlier discussion would also remind us that the abstraction will usually have side conditions attached to it; in this case the distinctness of the objects (no piles of sand) and their equivalence (all parsnips are equal).

The jump from recognising this similarity to developing the abstraction is not usually immediate: in many branches of mathematics this has taken hundreds of years. Usually the development of an appropriate abstraction is closely linked to the development of a suitable notation, although this may be misleading as the individuals may have used an abstraction, but not have been able to record the fact. In natural language we also see that the naming of a concept is closely linked to the understanding of it. Although it is possible to see similar situations and grasp those situations somehow as one, it is the naming of that new concept

which marks a watershed. Naming allows communication and is traditionally associated with power.

However, it is a mistake in searching for an understanding of a class of systems to jump into an abstraction prematurely. We have seen that there are many dangers associated with abstraction, and these are obviously heightened if we choose the wrong abstraction to start with. A better way to begin is to immerse oneself in the individual examples and look for the similarities. If something is a problem in one situation, rather than trying to generalise too soon it is often better to look at similar situations and ask what is the equivalent to the problem there. Looking at similar situations is a common experience: if we are faced with some new problem we will instantly think, "something happened just like that the other day". The analogy helps us to understand the new situation in terms of the old.

Abstraction and analogy can go hand in hand. If I say "the steel tube was as as cold as ice", the reference to "cold" tells us that it is the coldness (an abstraction) of the steel and the ice that is important; I would not expect there to be any similarity concerning the roundness of the tube or the wetness of the ice. In Chapter 6, the use of non-determinism helped us to see similarities between disparate types of problem. It focused us on certain aspects and thus acted as an abstraction of the different domains (sharing, real-time, uncertainty). The abstracted situations were not identical, but by losing some of the detail we were able to see that there were similar problems and thus similar strategies for dealing with them.

Analogies differ in their precision. Some are so precise that we can draw inferences about one situation from the analogous one. For instance, suppose I have two piles, one of oranges, the other of baked bean tins. I find I can draw a one-to-one correspondence between them perhaps by lining them up opposite each other; or possibly by painting coloured spots on them: one purple spotted orange, one purple spotted tin. I then take away one orange and one tin, and count the tins. If I have 57 baked bean tins left, I know I also have exactly 57 oranges.

Any similarity which is going to give rise to a formal abstraction has to be this precise. Mathematics is littered with words which describe such correspondences: translations, equivalences, congruences and morphisms of all hues (automorphisms, homeomorphisms, endo-, epi- and isomorphisms). Computing formalisms have added a few of their own, including bisimulation and observational equivalence. In music too (often allied with mathematics) we find transposition and counterpoint; however, the interest is often in the way that some theme is repeated in a similar, but not quite identical form.

Such looser similarities turn up in mathematics as well. Two mathematical structures may have a roughly similar structure, but differ in details. If some property is known to be true of the first structure, it is natural to wonder whether

a similar property holds for the latter. We may even try to follow the proof of the first in proving the other; these will differ from each other, and at some stage the analogy will break down and a different and more complex procedure will be required. However, we often find a similarity in gross structure.

This latter form of analogy is much more like the analogies found in literature and poetry. Simile and metaphor are some of the principal means by which poets convey meaning to their readers. Sometimes these can be pinned down, in a fairly tight manner, but sometimes are more suggestive. Let us look at probably the most well-known poem in the English language. Wordsworth says of the daffodils along the lake shore: (Hutchinson 1926)

Continuous as the stars that shine
And twinkle on the milky way,
They stretched in never-ending line
Along the margin of the bay:

Notice the level of detail in this simile. The milky way is continuous yet composed of distinct stars. The daffodils will similarly give the simultaneous appearance of a continuous mass and yet be composed of distinct flowers. The distinctiveness is especially obvious in their movement (they are "*Fluttering and dancing in the breeze*") and this is mirrored by the twinkling of the stars. Even the sweep of the milky way suggests the curvature of the bay. Of course, there is far more in such an analogy, subtle nuances, the cosmic nature of the milky way suffused with the magnificence of the panorama, but it is not too dissimilar to a mathematical analogy.

Of course, the PIE morphisms in §2.10 and the layered models of Chapter 7 are mathematical-style relationships in the context of particular models. When applying models, the most obvious "formal" method, direct refinement, is equally strong. We take a system, draw up a precise mapping to the model (this data type is the result R, this one the display D, these are the possible commands C, etc.) and then apply principles defined over the model to the system by direct translation.

In other places, we have dealt with slightly less precise analogies, more like the weaker mathematical analogies and perhaps the sort of simile above. We use the reader's understanding to draw relations between things that are not 100% precise, but yet which carry an obvious meaning. The reason that we went into so much detail about the simple PIE model in Chapter 2 was to perform various analyses on this simple model, so that they would not need to be repeated later. The implication was that although the models were slightly different there would be a roughly similar behaviour. For instance, consider the definition and construction of the monotone closure (§2.5). This cannot be applied directly to the windowing model, or the non-deterministic PIE, but something like it would apply. We would certainly require a more complex definition in each of these cases, but would retain the general notion of a state which captures just what

affects the user, but no more. Further, the formal definitions would be very like the definition for the PIE, but have extra bits and small differences.

In Chapter 6, we developed a non-deterministic version of the PIE model. It is fairly clear that this process could be applied to other models. Similarly, we have seen properties of different models labelled as forms of predictability, observability or reachability, and aliasing has cropped up in different forms. In each of these cases the analogy is not quite precise, but allows us to organise and understand the relevant formal models.

Think back to the lift example in §11.2. The analysis there *could* be seen as completely formal: the command set is the lift buttons, the display consists of the various lights inside, etc. However, my *initial* analysis was more one of simile. It was the "likeness" to the model rather than the precise relationship which sprung to mind. Indeed, to complete the formal analysis of the lift system I would have needed a model which dealt adequately with the external aspects of the system such as the actual movement of the lift and other passengers. Note then that I was able to draw the formal relationship only because I had first seen the informal analogy.

Let's go back to the poem. It begins:

I wandered lonely as a cloud
That floats on high o'er vales and hills,

At first sight this is like the "cold as ice" analogy. We are comparing Wordsworth to a cloud and it is the abstraction of loneliness which is of interest. However, a moment's thought and we realise that clouds are not lonely. The cloud is alone surely, but lonely? It is only by personifying clouds that the analogy makes sense. We see Wordsworth in the cloud, then feel the feelings he would feel. The second line unfolds this a little. The cloud floats high above the earth; Wordsworth, although physically upon the ground; is wandering and abstracted. The cloud's floating is like the floating of a lonely person, like Wordsworth himself. The simile tells us as much about clouds as it does about Wordsworth.

A moment later Wordsworth sees

..... a crowd,
A host of golden daffodils;

This is terse metaphor. We could read it as a very precise analogy: a "crowd of daffodils" means a lot of daffodils close together. However, because Wordsworth was lonely and alone, the word crowd evokes all sorts of additional feelings: companionship, togetherness. The contrast is especially pertinent when we compare the floating clouds that do not touch the ground, to the daffodils which are gaily rooted and part of the lakeland landscape. Again, the analogy has a richness far beyond the simple matching of attributes.

Let's look again at abstract modelling. In Chapter 6, we developed a non-deterministic model to help us deal with some formal properties of systems. We wondered what this signified in the real world. Now some of the implications were direct formal analogies between the models and the real systems they denoted. For instance, non-determinism due to timing corresponds exactly to the non-determinism that arises in the formal model. However, the discussion ranged far wider than that. The analogy between formal models and interactive systems had a richness, which although by no means as beautiful is not so dissimilar from the poetic analogy.

The discussion of events and status in Chapter 10, although rooted at various places in formal models, ranged far wider than these models. Formal and informal concepts were counterpoised, and, like Wordsworth and the cloud, we ended up knowing more about both.

This stretching of understanding generated by the use of formal models is possibly their greatest benefit. The application by strict refinement is necessary and perhaps desirable in many situations, but is rather utilitarian compared to rich and exciting formal metaphors. Not only can I do more, I know more. Nowadays when I see a system, one of the things I look for is aliasing. Note I do not match the system to a model and then look for aliasing via the model. Aliasing has become part of my understanding. The same could be said for other properties I have mentioned, such as predictability, dynamism and structural change.

I am not sure how many authors would admit it, but I believe that many uses of formalism are of this analogous nature. We have already discussed the way that formal statements often serve much the same purpose as box diagrams, and often less clearly. There are circumstances, especially where temporal reasoning and change is involved, where diagrams are not so useful. In these cases a formal model may be enlightening even if it is not an accurate reflection of the real system. This is because it serves as an analogy. Problems arise if the author mistakenly believes that the formal statement is in precise mathematical correspondence rather than analogous to the real system. There is nothing wrong in telling a reader exactly what are the limits of a particular statement, but go on to use it to guide an analysis of a wider class. However, to go on in ignorance may be dangerous.

Of course, this discussion of poetic analogy and formal analogy is an analogy in itself. It would be unwise to be too precise in looking for a correspondence between Wordsworth's poems and abstract models. However, as a looser, more poetic analogy itself, it suggests some of the richness that we can gain through analogous use of formalisms.

12.4 Other themes

This chapter has been concentrating on abstraction as a unifying theme. However, a few other concepts have repeatedly arisen during its course which deserve reminders.

Some of the earliest properties we discussed concerned the *predictability* and *reachability* of the system; or to put it in other words, what you can see and what you can do. Similar properties have recurred in subsequent chapters, especially various forms of predictability. For instance, in Chapter 5 we wanted to be able to observe the current state of systems liable to delayed responses or buffered inputs. Later, in Chapter 9, we were particularly interested in predicting the effect of view updates and whether or not an attempted view update would succeed. The ability to know what is and will happen is an obviously desirable property of a system. Chapter 6 reminded us that what is or is not predictable depends on who you are. Some response may be predictable to the program's designer, but not to the user; to the expert, but not the novice. If we detect such non-determinism, we can begin to think about ways of making such a system more predictable.

The way predictability was defined for PIEs in Chapter 2 emphasised the possible future interactions with the user. In Chapter 3, we found that predictability and observability properties were expressed most easily as functions denoting what could be deduced from one view about another. The fullest form of predictability was when we could tell the whole current state from the display or collection of displays. It was important that this state was not the explicit internal state, but the *monotone closure*, the state as it may affect the user. This same insistence on not looking at the explicit state as given by the implementation arose again in Chapter 4. It was important there that any definition of sharing or interference dealt with the implicit linkage as perceived by the user, rather than any explicit data model.

Connected to, or perhaps a special case of, predictability was the issue of *aliasing*. This has cropped up already in this chapter, and we saw examples of aliasing in Chapter 1, where the concept was first introduced, Chapter 4, where we considered aliasing of windows, and Chapter 9, when we asked whether users were aware of what views they were seeing. Aliasing is an important special case of predictability because it is easy to overlook. When assessing a design, we will more easily notice that certain attributes of an object are not visible, than that the objects are not themselves uniquely identifiable.

In several places there has been a stress on *dynamism*. This was most obvious in Chapter 8, where we discussed dynamic pointers as compared to their static counterparts. Also it was a key feature in the distinction between status and events in Chapter 10. Indeed, the relationship between events and dynamism

was the focus of part of the summing up of that chapter. Because of the major differences between static and dynamic properties of an interface, Chapters 6 and 7 attempted to divide the consistency properties into static and dynamic invariants. However, this was mainly in order to emphasise the special role of static invariants which are applied before the dynamic ones. This seems to contradict the stress on dynamism, but in fact highlights a nice counterpoint between the two. Static properties tend to be what make an interface correct, whereas dynamic ones tend to make it interesting.

One form of dynamism which is particularly important is *structural change*. This underlies the requirement for dynamic pointers; if the data structures were structurally static the pointers to them could be, and hence we would require no *pull* functions, and everything would be a lot simpler (but less fun). Although we discussed some properties of windowed systems when windows were being created and destroyed (§4.8), the required properties are far less obvious. Similarly, when we thought about dynamic views at the end of Chapter 9, these clearly had much more complex properties than the static case. Note, we had several levels of dynamism going on here. The simplest case would be simple structurally static views: the views are simply what we see and we never try to *do* anything through them. The chapter was principally about updating through views. We were thus interested in an aspect of dynamism: the contents of the database changed. However, in so far as it affected the view we assumed that the database was structurally static. We then considered the case where the views "moved" when the database was altered. We noted how dynamic block pointers could form a descriptive or implementation mechanism, underlying the importance of dynamic pointers. The final case, which we considered only briefly was when we wanted to update the fundamental structure of the data through views. Each level of dynamism added complexity. In the introduction to Chapter 5, we noted how the temporal properties of interfaces are often poorly specified and documented. One simple reason for this is that it is easy to draw a sketch of a snapshot of a system, but more difficult to describe its evolution. Precisely because it is easy to overlook and complex to handle, we must be especially vigilant when considering aspects of structural change.

Having talked about structural change we move on to *structural correlation*. Precisely because structural change is difficult to handle we want to avoid it in the design process. As far as possible we want the design notations, models and specifications to match the designer's intuitions. There is an obvious tendency to specify systems in an easily implementable fashion. This is correct up to a point: it is no good designing a wonderful interface only to find that it is totally impractical. However, this process of checking the realisability of a specification should not dominate the structure. As I pointed out right at the beginning of this book, the greatest barrier to a successful interface is the *formality gap*. Our major effort in design must go towards narrowing that gap as much as possible.

This might mean defining new models or notations which match the domain, or simply choosing appropriate structures within the notations we have. Structural transformation within the formal domain might be a bit of a pain, but is fundamentally manageable. Indeed, we saw how *interface drift* allowed us to perform structural transformation even in the presence of conflicting goals.

12.5 Ongoing work and future directions

The work described in this book is not static. I will now describe briefly some more recent related work and possible future avenues for development.

I have shied away rather from making the user an explicit part of the model, preferring the psychological insight to be in the choice of level of abstraction. I have given some reasons for this reticence above (§12.3.2), but within these constraints there is room to make the cognitive elements more precise. I have already noted the work on display and result templates (Harrison *et al*. 1989). These try to add a more psychological perspective to the display and result by adding an idea of focus. This still leaves the defining of such areas of focus to the human factors expert or to experiment, and therefore does not run the risk of reducing the user to a model. Perhaps other cognitive aspects could be included explicitly into models without compromising the user's humanity.

Another feature of these studies is the way that they developed a specific model, the *cycle* model, to capture aspects of a particular system. They were studying a specific bibliographic database system, but found it useful to abstract away from the specifics of the system in order to understand it better. This model was not as wide in its generality as say the PIE model, but was better able to express specific properties of interest, in the same way as the model in Chapter 4 addressed windowed systems. This process of finding domain-specific *interaction models* has been promoted to a specific design and analysis heuristic.

Most of the models in this book are largely *declarative* in nature. They are intended to describe systems in such a way as to make it easy to define properties but not to determine how those systems work. This is the correct first step: the discussion of the formality gap told us that we must concentrate first and foremost on *what* we want to be true of the interface. However, we saw in §11.5 that we cannot expect the structure of such a model to match the structure of the implemented system. More important, the definitional models need not even be of suitable structure for the specification. The models address specific issues, and ignore others, but a complete interface specification must cover all areas. There is thus a need for parallel *constructive* methods which are tied into the individual methods. We could just take an existing specification notation and map the model into it, but this would not be too successful without some sort of methodological guide; the gap is too big. The specification in §11.4 used layered

models and pointer spaces to generate an architectural plan. These two formulations are themselves examples of intermediate formalisms. I regard dynamic pointers as a central area for study and experiment, partly because they crop up so often, and partly because they are useful in implementation as well as definition. Sufrin and He's (1989) interactive processes and Abowd's (1990) agent architectures are also aimed at this gap.

The big gain from developing generic architectures is that a lot of the work of checking properties can be done as general proofs at the abstract level; specific uses of the architecture would only require the verification of simpler properties. In practice, however, I think that the complexity of real systems means that more domain-specific models are often required. These are able to have more of the usability prepackaged, and come with simple rules for building, but are limited in their scope.

In the spirit of this chapter, I don't want to finish on the formal application of formal systems. Some while ago Harold Thimbleby and I were discussing consistency in interfaces. Metaphors like the desktop were clearly very useful, but tended to break down when pushed. In general, we concluded, you could not expect powerful complex systems to be expressed entirely by simple direct-manipulation interfaces. However, we began to use the analogy of differential geometry (the underlying mathematics of Einstein's General Relativity). We imagined systems which were too complex to be represented by a single simple interface but which were composed of simple overlapping parts; where each part had a direct manipulation interface and where the boundaries between these parts were consistent. I don't vouch for the practicality of the idea, but we found that the formalism was a tool for thinking, a spark for the imagination, and fun.