CHAPTER 9

Complementary functions and complementary views

9.1 Introduction

At the end of Chapter 7, we took two branches. One was based on the "editing the object" approach and was followed up in Chapter 8. This essentially took the view that all our update commands are addressed to the underlying object and the display acts merely as a way of viewing the effects of those updates. The other branch was the "editing the view approach" and assumes that it is the user's view of the object that is primary. Updates are then addressed to the view, and the underlying object must keep pace with these changes. This is arguably a more user-centred approach, and is investigated in this chapter.

I say arguably because if we are to have a good user interface the two approaches will converge. On the side of editing the object, the presentation attempts to uncover the object's structure on the display. On the side of editing the view, if the global effects of updates to the display are to be predictable, then again the map between object and display and the effects of updates must be transparent. Good design will probably consist of a blend of the two approaches, and it is hardly surprising that at the end of this chapter we end up using the pointer space designed out of the former strategy to model structures resulting out of the latter.

In the rest of this section, we introduce the concept of a *complementary function*, which acts as a unifying concept for several problems in addition to editing the display. The problem addressed by the rest of this chapter can then be seen as one of finding suitable translations from update functions to their complements. We then look at some of the principles that we will wish to apply to translation strategies. In particular, predictability will play a pivotal role in driving the chapter.

Researchers into database theory have already investigated this problem and we use their terminology. However, we will see that the databases normal to this field differ in several respects from those common in general interface design. That is, our structures are both *dynamic,* and subject to *structural change*. Despite this, we look principally at the case of static views, as these are far more tractable and at least give us the general flavour required for more complex cases. The databases with which we deal differ markedly from typical DP databases in their structural complexity and scale. For instance, we may be interested in program syntax trees, or a small set of numbers, rather than large homogeneous relations. This is particularly important since many of the views defined over databases are slices put together out of these large relations, whereas those in which we are interested may be far more complex (even when static).

The first major section gives a short algebra of views, so as not to clutter up the succeeding sections with too many definitions. Of particular importance are the definitions of *complementary* and *independent* views. The reader could choose to skip this and refer to it as necessary.

Section §9.3 reviews possible translation strategies, starting with a rough taxonomy of strategies based on the level of generality at which they operate. It then examines the advantages and disadvantages of *ad hoc* translation schemes and schemes based around product databases. Finding these wanting, we go on to discuss the use of a complementary view which stays constant as the view of interest is updated. This is seen to have many advantages and bridges the gap between totally *ad hoc* methods and the restraints of product database formulations.

Although the use of complementary views allows a level of predictability over the updates to the underlying data, it has unpredictable failure semantics, and §9.4 introduces the new concept of a *covering view* that supplies sufficient information to predict failure. The security implications of this are discussed.

9.1.1 Complementary functions

In addition to the problem of editing an object through its display, there are several other problems which, although differing markedly in some respects, all share a common structure:

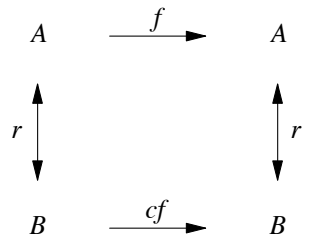
- We are viewing a representation of an object on a display and then change it (perhaps insert a character into a text file). We wish to update the display image by making small changes rather than completely recreating the picture.
- We are editing some view of a large database; we wish the underlying database to change in a way consistent with the changes in the view. We will see that this is usually far simpler than editing the display.

- We want to say that the changes to an object are confined to a locality.
- We are editing several objects, all of which must remain consistent with one another. If one is changed, the rest must change to remain consistent.

Each of the above problems reduces to the following: we have two sets (A, B) representing the objects/views and a relation (r) between them; we have a transformation function $f : A \rightarrow A$ and wish to find a *complementary* function, $cf : B \rightarrow B$, which is consistent in that:

$$a \leftarrow r \rightarrow b \quad \Leftrightarrow \quad f(a) \leftarrow r \rightarrow cf(b)$$

That is, the following commutes:



In cases (i) and (ii) the relation will often be directed: in case (i) A is the object, B the view and r is the viewing function $A \rightarrow B$; in case (ii) the function is the other way round.

We could call this the "spider's legs" problem. The legs represent the various views and they are joined by the body (the relation): as we pull one leg the others must follow. Note that this is equivalent to the problem of polymodality in logic programming. The finding of a complementary function is sometimes called translation (Bancilhon and Spyratos 1981).

9.1.2 Principles

Several problems may occur in defining such a translation scheme:

- *Uniqueness* – Unless the relation is functional $A \rightarrow B$ there will be many choices of cf for a given f.
- *Failure* – Again, unless r is a function $A \rightarrow B$ there may be *no* complementary function possible that satisfies the constraints.

Similarly, when we have chosen a particular translation scheme, we still have the same problems in new guises:

- *Uniqueness* – If there are several updates that achieve the same effect on A, do their complementary functions achieve the same effect on B?

That is, for any pair of updates f, g and objects a, b such that $a \leftarrow r \rightarrow b$, is it true that:

$$f(a) = g(a) \quad \Rightarrow \quad cf(b) = cg(b)$$

- *Failure* – Even where there is a consistent choice of a complementary function, it may not be defined for a particular translation scheme. For example, we may disallow the editing of key fields in a record, perhaps forcing the user to delete and re-enter such records. Further, for a particular update the complementary function may be partial. We may allow the editing of the department field of an employee record, but insist that the new department should already exist, even though in principle it would be possible to create a new department spontaneously.

We can relate these issues to the usability principles of predictability and reachability (Chapter 3) and to sharing properties (Chapter 4):

- *Predictability* – Does the user know what translation scheme has been chosen? Does the scheme admit a model by which the user can infer its behaviour? Can the user predict whether a given update will *fail*? If not, what additional information is required?
- *Reachability* – The choice of translation scheme will affect the updates that are possible to B via the complementary functions, and also, because of the choice of failed updates, those that are possible to A.
- *Sharing* – In the case of several views being edited, the choice of the translation scheme will affect sharing properties such as the commutativity of updates on different views.
- *Aliasing* – Do the users know what they are seeing? That is, do they know what the viewing function is? If they do not, there is no way they can relate what is seen and manipulated in the view to the underlying state of the system.

9.1.3 Complementary views – what you can't see

Aliasing reminds us that it is important that the designer and user are agreed as to what is in a view. For example, my bank statement has a running account balance. My interpretation of this view is how much money I have got, so if the amount is always positive I think I am in the black. However, the bank regards monies from cheque deposits as being available only when the cheque has cleared, and thus although the amounts shown on my statement are always positive I get charged interest and transaction charges.

The problem is partly one of observability: the statement does not tell me my available balance, and thus does not tell be whether I am "really" in credit. To obtain this information from the statement means I have to remember what

proportion of my deposits are cash and which are from cheques. It is also an aliasing problem, as a bank statement based on the actual date of deposits and one based on available balance would both appear the same. That is, the *content* of the view does not tell us about the *identity*.

Aliasing is a thorny problem and may only become apparent when a breakdown occurs (as when I was charged on my account). However, the designer and the user do at least have the common view before them and this can serve as a focus for thrashing out a common interpretation.

We can characterise the problem of aliasing as *knowing what you can see*. A new issue that will arise as the chapter progresses is the importance of *knowing what you can* **not** *see*. Whereas for observability the important issue for designer and user is agreeing on what they do see, the crucial issue for update semantics is agreeing on what they do not see.

This seems a rather surprising statement, but the thing that makes it possible to understand change is knowing that certain other things do not change. When we edit one file, we expect other files to remain unchanged. When we pull the plug out of the bath, we expect the water to stay in the sink. So, for each view we will look for a *complementary view* which remains constant through updates. It is hard enough to get over the problem of aliasing. The designer and the user have trouble agreeing on what they *can* see. It is clearly even more important that explicit effort is made in the design and documentation of systems so that the user knows about this often implicit but crucially important view. Knowing what stays constant, i.e. knowing what you do not see, is fundamental.

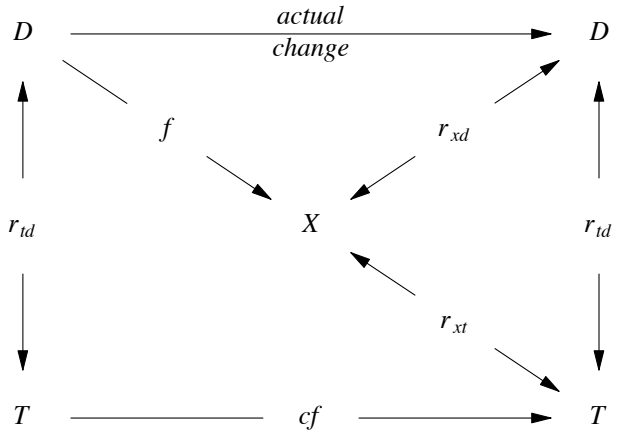
9.1.4 Extra complexity – the text editor

The data handled by typical interactive systems turn out to be far more complex than a typical DP database. This is the case even for apparently simple data structures such as the text of a document. We'll take a look at a text editor and the choice of complementary function for the two modalities.

The text consists of an unbounded number of lines all of the same fixed length (for simplicity) and a cursor position within the text. The display consists of a fixed number of lines of the same length as the text and a cursor also. The invariant relation will be that the display should be identical to the portion of text it would cover if the cursors were lined up.

- *Editing the text, display following* – Take, for instance, inserting a new blank line after the cursor. We are then faced with problem of non-uniqueness: should we lose the top line of the display, or the bottom? Usually the choice is different depending on the position of the cursor in the display.

- *Editing the display, text following* – This case is more difficult. If we again consider inserting a line on the screen, and say the bottom line of the screen is scrolled away, would we expect it also to disappear from the text? If not, why not? It would certainly be consistent with the invariant: obviously, some choices of complementary function are better than others! Even more bothersome is the case of deleting a line. Usually a new line "appears" from the hidden text to fill the gap. We are hardly then editing the view. In fact, to describe this adequately (and line insertion), we need a slightly more complex idea of complementary function. When we insert into our (say) 80x25 screen we obtain a 80x26 screen, and deletion similarly gives an 80x24 screen; we then expect these to bear a relation to the actual new display. Thus our screen update is from the real display (D) to some sort of extended display (X). We get the following diagram:



Given such complexity, is editing the view, rather than the object worth thinking about? For simple cases (like text!), probably not, as it is reasonable to expect the users to perceive the underlying object as the item with which they are interacting. However, as the underlying object and the map between it and the view get more complex, it becomes more and more likely that the user will manipulate a surface model. For instance, when editing a file one rarely thinks in terms of the reads and writes to the disk and the allocation algorithms for disk space that are going on underneath; the file system is just such a complex view of the disk.

One other thing that is worth noting from the example is that we tend to think of the relation between text and display as functional: "this display is the text between the 53rd line and the 77th". This is a valid perspective and the one we

will use for much of this chapter. However, in the case of the display of a text, the view moves. We want dynamic or semantic views, not static ones. Fortunately, in many situations the forms of view do not require this complexity.

9.1.5 Differences from standard databases

The example above shows that we should not expect all the properties of DP-style databases to hold for the more general case. In most DP situations, the structure of the database is fixed for long periods of time, and changing that structure is seen as a major, expert task. If we consider general systems for which we may want to design an interface, this may not hold. For instance, we may want to design an interface for manipulating a file system. We clearly want to create and delete files as well as edit them.

In addition to this structural change, we also may have far greater structural complexity. So for instance, a portion of a program syntax tree is far more complicated than a list of personnel records.

9.1.6 Summary

The concept of a complementary function over a relation captures many diverse problems, but we here confine ourselves to the case where the relation r is a function and usually refer to it as v, the principal view.

We must be prepared to examine the cases of dynamic, structurally changing and structurally complex views. However, we initially look at the simple case of possibly complex but static views. Further, we assume until §9.5 that the database is structurally static. This means that all views are valid for all database states.

9.2 Algebra of views

The rest of this chapter considers complementary functions on databases resulting from updates to views. This section gives a short exposition of those aspects of the algebra of database views that will be needed later. The definitions are not new; they can be found elsewhere (Bancilhon and Spyratos 1981).

A view is regarded simply as a function on a database, v: $Db \rightarrow$ **range**. If the view is being used to produce some sort of output for the user, then the form of the map will be very important. However, as in the case of the definitions over PIEs, we do not consider fidelity conditions here. Other views will be wanted for their conceptual or definitional value, and are not intended to represent a user image. In these cases fidelity will be irrelevant and only the information content of the view is important. Hence two views would be equivalent if they had the

same information content.

9.2.1 Information lattice

Where we are interested only in the information content of views, we can define an information ordering on them. One view is considered a subview of another if the former can be derived from the latter:

{ *subview* }:
u is a subview of v, $u \leq v$ **iff**:
$\exists g$: **range** $\rightarrow$ **range** **st** $\forall db$ $g \circ v(db) = u(db)$

As when we considered PIEs, definitions concerning information can be made using functions (as above) or more implicitly:

$$u \leq v \quad \equiv \quad \forall db, db' \quad v(db) = v(db') \quad \Rightarrow \quad u(db) = u(db')$$

That is, we can always work out what is in view u if we know what is in view v.

This clearly gives a partial order to the views. This partial order admits a least upper bound or supremum by using the product view:

{ *product* }:
$(u \times v) : Db \rightarrow$ **range** $\times$ **range**

$(u \times v)(db) \quad = \quad (u(db), v(db))$

This is clearly the minimal view (with respect to $\leq$) of which both u and v are subviews.

If the views are for the user, then this corresponds to a new view where both the originals are visible. So it would still be the least upper bound even if we added quite strict fidelity rules on the information order.

There is also a maximal element, amely the identity function on the database, id_{Db}. In both the case of the supremum and the maximal element, whether they are acceptable or not as a "real" visible view depends on the set of views regarded as acceptable. In the case of normal databases the former is probably acceptable, whereas the latter, implying that the user can see all the database at once, is arguably not reasonable. For the case of display views of texts, even the supremum represents a display twice as large as is possible. However, we only need to use the supremum to make definitions, so it can be used even when unacceptable as a normal view.

9.2.2 Independent views

It will be important later to know whether two views are interrelated or not, that is, whether the views can vary independently of one another. We can define independence of views:

{ *independence* }:
u,v are independent **iff**:
$\forall a \in$ **range**, $b \in$ **range**
$\exists db$ **st** $a = u(db)$ **and** $b = v(db)$

That is, for any given pair of possible values for u and v, we can find a database simultaneously yielding those views.

As well as using this concept later when we consider translations of views, it is clearly important when we consider sharing. For example, consider two views being presented in two windows. The database is being manipulated by editing the views and we assume all updates to the views are possible. Clearly, the two windows cannot be display independent unless the views are. If the views were not independent then there would be circumstances when a change in one view would yield a situation where there would be no consistent database state yielding both views.

As in the case of display independence in Chapter 3 and independence of blocks in the previous chapter, pairwise independence does not imply independence of more complex configurations. For example, consider a database consisting of three numbers, a stock level, an amount on order, and a minimum stocking level:

$$db = stock\_level, amt\_ordered, min\_stock$$

There is the sensible constraint that the actual stock level plus the amount on order must always exceed the minimum stock level:

$$stock\_level + amt\_ordered \geq min\_stock$$

If we consider the three views giving the three numbers separately, then each view is independent of the other two;‘ however, they are clearly not independent as a group. Thus in this example (and in general):

u independent of v, **and** u independent of w
does **not** imply u independent of $v \times w$

Similarly, we could create arbitrarily complex situations with n-way independence but not $n + 1$-way independence.

In this example, we could have "normalised" the views, to make them independent, by considering the view set:

$$\{\ stock\_level,\ amt\_ordered,\ stock\_level + amt\_ordered - min\_stock\ \}$$

These views are independent in all configurations, but the last view is, of course, by no means sensible. For example, updating the stock level when an item was sold would implicitly reduce the minimum stock level! This example gives us some inkling of the complexity of view editing.

9.2.3 Complementary views

In terms of the information lattice, the opposite of two views being independent is their being *complementary.* That is, together they contain all the information in the database:

{ *complementary views* }:
u and v are complementary **iff**:
$\forall db, db'\quad u(db) = u(db')$ **and** $v(db) = v(db') \implies db = db'$

or, in terms of the supremum:

$$(\ u \times v\ ) \quad \equiv \quad id_{Db}$$

A third, functional formulation is:

$$\exists f\colon \textbf{range} \times \textbf{range} \to Db \quad \textbf{st} \quad \forall db \quad f(u(db), v(db)) = db$$

The function f is, of course, unique and is the inverse to the supremum, so we could say:

u and v are complementary **iff** $(\ u \times v\ )^{-1}$ exists

In general there may be many views complementary to a given view. Consider a database consisting of two integers, x and y, constrained so that $x > y$. If we have one view giving x (call it X), then the view giving the value of y is complementary to X, but then so is the view giving y cubed. These two views contain exactly the same information so could be said to be equivalent. But even if we consider information content, the view yielding $x + y$ is also a complement to X but is different from Y. Further, we may add information to either of the views, and they remain complements.

In fact, view independence and complementariness satisfy the obvious conservation properties with respect to subviews:

if $u \leq u'$ **and** $v \leq v'$
then
u' and v' independent $\Rightarrow$ u and v independent
and
u and v complementary $\Rightarrow$ u' and v' complementary

9.2.4 Summary

In summary, we have defined four basic concepts:

- *Information or subview ordering* – This says when one view contains enough information to determine another.
- *Supremum over this ordering* – This yields a view ($u \times v$) containing as much information as u and v put together.
- *Independence* – This tells us when two views can take any pair of values.
- *Complementary views* – This is when two views are sufficient to determine the entire database and, in particular, implies that there is an inverse to the supremum.

9.3 Translation techniques – complementary views

This section reviews different ways of translating updates to views into updates on the underlying database. It starts off by recalling the translation problem, and putting it in terms of the notation that is used in the section. This is followed by a subsection which classifies translation strategies into different levels.

Sections §9.3.3, §9.3.4 and §9.3.5 deal in turn with *ad hoc* translation, translation when the underlying database can be regarded as a Cartesian product of views, and finally, translation using a complementary view.

9.3.1 The view update problem – requirements

To recap the situation, we are concentrating on a view v on a database Db. We wish to find a complementary function for any updating function f on the view. In the terms used by the database community, this process of finding the complementary function is called *translation*. Thus the problem which we address is:

{ *translation* }:
given a view v and a function f: **range** → **range**,
find a translation $Tf : Db \rightarrow Db$ such that:

$$f \circ v \quad = \quad v \circ Tf$$

If we think of the view update as giving us an explicit change for a part of *Db*, the translation problem can be seen as determining what happens to the things that are not mentioned explicitly in the view, or are specified only partly by the view update. It is thus a form of the *framing* problem.

9.3.2 Translation taxonomy

Two major determinants of the update strategy are:

(i) What update operations are available as translations?

(ii) How much information about the update is required to calculate the translation?

Dealing with point (i) first of all, it is clear that there may be limitations on the sorts of updates that are possible. For instance, in the case of updating a display in line with a text, there are the available terminal codes, and in the case of a view on a database, there may only be certain operations allowed on the data by its underlying implementation. It will often be assumed that any transition between consistent states is possible. This assumption may or may not be reasonable. On the positive side, we must assume the set of updates available are complete, in that any consistent state can be obtained via some series of updates; if this is not so then we must assume there is something seriously wrong with our implementation. It is fundamentally unreachable. On the negative side, even though in principle such a sequence exists, it may be arbitrarily complex and therefore hard to generate. Such complexity is potentially disastrous in a shared database, where a large part of the data may be locked during a very long transaction. An important practical consideration for any translation scheme, is that the complexity of the update should be of the same order as the complexity of the view function.

Considering (ii), we need to clarify the level of instantiation at which we wish translate. Typically, updates are parameterised functions, for example:

"insert the character _?_ into the text"
"change date to __?__"
"delete the record with key __?__"

Thus the general update is of the form:-

level 1 – "add record": $param \times A \rightarrow A$

We can then instantiate this for a given parameter:

level 2 – "add record *Fred born on 29/2/60* ": $A \rightarrow A$

and finally, for a given view instance, $a \in A$:

level 3 –

birthdays	
NAME	DATE
Jill	3/7/53
Tom	27/10/61
Glenda	15/4/47

$\rightarrow$

birthdays'	
NAME	DATE
Jill	3/7/53
Fred	29/2/60
Tom	27/10/61
Glenda	15/4/47

In the expression of the translation problem, we have dealt implicitly with level 2; however, if the translation function *Tf* is not a basic update operation on *Db* but instead does different updates depending on the the exact value of the view, then it is operating at level 3. On the other hand, if it is generic over a parameterised class of update functions then it operates at level 1.

When a translation scheme operates at level 1 or 2 , giving for each update operation a fixed sequence of database updates irrespective of the database state, we can say it is *context independent*. For example, when keeping a text in line with a screen display, many operations yield context-independent updates; so that "insert _?_" at the cursor point on the display translates to "insert _?_" at the text cursor, irrespective of the current text and display states.

On the other hand, some translators at level 3 may need the value of the view to decide on the update, but may not need to know the particular update function (e.g. *add_record*) used. That is, they have the property that they are not dependent on the *type* of the update, only on the previous and updated states. We could call such translations *process independent*. An example of this would be files as views of a file system. When an edit is finished, the update to the file system is dependent only on the final value of the edited file, not on the particular edits which were done to the file.

Other translations will be hybrid: for instance, when keeping a screen display in line with a changing text, the update "delete the line below the cursor" may become something like:

> "delete the line below the cursor,
> then :
> if the cursor is near the bottom of the screen add the relevant line of the text to the bottom of the screen,
> otherwise add the relevant line of the text at the top."

It should be noted that a translation may be either context or process independent at the behavioural level, yet have an implementation that does not posses this property. This is particularly the case if the update strategy is defined using operations that are not available on the database directly, but have to be simulated: in particular if one assumes free update of the database.

9.3.3 Ad hoc translation

We can approach each task of update translation in a one-off way. In the short-run, this is the simplest translation strategy. For instance, we might supply an index of variables used in a program:

```
amount      occurs 2 times
    amount  is static variable in main
            declared line 3
            used lines 7 and 8
    amount  is parameter to do_work
            declared line 73
            not used
total       occurs 1 time
    total   is static variable in do_work
            declared line 77
            used lines 86, 87, 93 and 101
```

We begin to ask what updating various fields might mean. We decide that updating the variable name `amount` in the "`occurs`" line would be regarded as a global renaming of all occurrences of the variable name. However, it would have to fail if this resulted in any name clashes. Similarly, we might want to allow editing of the variable name in the "`is static in main`" line. This would rename the variable just in that context; again it might have name clash problems, but would also mean moving this to a new subheading. Editing the procedure name within which a variable is used could be interpreted as a renaming of the procedure; however, editing the line numbers where a variable is referenced makes no sense at all.

Even where updates are allowed, we would need to be careful. For instance, if we wished to edit the variable `amount` to read `total` we would have an intermediate state which would lead to name clashes. The edits would therefore have to be committed in non-atomic units.

Clearly it is a complex job deciding what is updatable, and what those updates mean. Features of translation that are apparent here are:

- Only a subset of possible update operations are allowed. Further, this example has assumed that updates are the natural ones for the view. In

fact, the set of updates will usually be defined more rigidly by the underlying data objects and are not necessarily sensible ones for the view.

- Those updates that are allowed may fail sometimes. (This is a feature of all translation strategies, and of normal updates.)
- The semantics of updates may have to be changed radically: for instance, editing a variable name in an individual use would relocate that use to a different variable heading. (Again, this is not solely a problem of *ad hoc* strategies.)

Ad hoc strategies have several disadvantages:

- *Predictability* – It may not be clear to the user what the effect of an update will be. In particular, it may be that different ways of achieving the same change in the view may have different effects in the underlying database.
- *Reachability* – It may be very unclear what changes can be achieved through the view.
- *Dishonesty* – It is easy to fool oneself and others that it is the view that is being edited, whereas this is really a subterfuge for editing operations on the underlying object. This is particularly a problem if the concentration is not on "what would editing this value mean", as in the example above, but instead on "what can we do with database updates".

Perhaps the most important of the criticisms above is the predictability of the consequent changes in the unviewed parts of the database. For instance, in our stock control example, it would be sensible for any update of stock level to generate an order and hence change the number on order, but not reduce the minimum stock level. Making this sort of inference in an *ad hoc* system can be difficult.

On the other hand, an *ad hoc* approach has important advantages:

- *Generality* – There are no fixed limitations to the sort of view we can tackle. But it may be difficult!
- *Task orientation* – It is designed for a particular task and can be fitted to it. For instance, it may be deemed inappropriate to edit procedure names via a variable names index. This form of task-specific limitation may be difficult to achieve using a more unified strategy, and even where it is possible, it is less likely to be noticed.

Because of these disadvantages, and in spite of these advantages, we now look at more unified and generic approaches to translation.

9.3.4 Product databases

The simplest case of view update is when Db can be expressed as a Cartesian product and the views as "slices" of this. That is, $Db = \prod_{\mu \in M} A_\mu$ and:

$$\forall v \in V \quad \exists \mu_1, \cdots, \mu_n \ \textbf{st} \quad \textbf{range} = \prod A_{\mu_i} \quad \textbf{and} \quad v(< a_\mu >_{\mu \in M}) = < a_{\mu_i} >_{i \in \{1, \cdots, n\}}$$

In this case, for any given v we decompose Db into a product $A \times B$ where A is the product of the A_μ "sliced" by v and B is the rest. Then we have simply $v(< a, b >) = a$ and for any update $f : a \rightarrow a'$ we have a translation to $Tf : < a, b > \rightarrow < a', b >$. A simple (flat) file system is an example of a product database where the individual files are the views.

If the situation is more complex and there are invariants to be preserved of the tuple, then it is tempting simply to amend this so that for any consistent pair $< a, b >$ the update $f : a \rightarrow a'$ has a translation $Tf : < a, b > \rightarrow < a', b >$ if the latter is consistent, otherwise it fails. This strategy works quite well if the consistency relation is simple.

An example of this is a relational database with the views being the relations. If we can normalise the relation so that the only consistency requirements are those of non-null key fields and existent or null alien keys, then the former is within a single relation and the latter is the only cross-relation (and hence cross-view) constraint on update. The reason why this particular relation is acceptable is that the reachability is preserved; in order to delete a tuple, we can simply go round the views nulling all references to it, and then remove it. Similarly, to add a set of mutually referencing tuples we can add them with nulled alien keys and then go round filling them in when all the tuples are in place. Not all cases are so simple, and it is easy to think of consistency requirements that totally disallow certain updates. One of the aims of normalisation is precisely to make these consistency requirements simple, and to push as much semantics as possible into the relational structure.

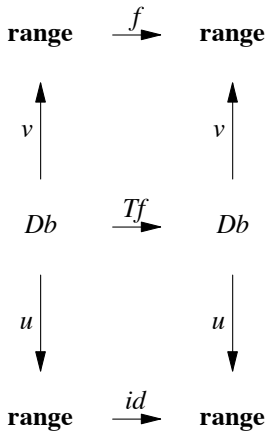
In principle, any database with views could be regarded as the Cartesian product of those views with suitable consistency requirements, but unless these requirements are very simple (as above) there is not necessarily much to be gained in terms of update specification. The approach taken by Khosia *et al.* (1986) is considerably more complex, being based on modal logic. However, their view that a database should be specified first as a collection of views, has some of the characteristics of a constrained Cartesian product. Their update rules between views are *ad hoc*.

9.3.5 Complementary views

The general idea that, for a *particular* view, we can decompose the database into a product is more useful. In fact, the attempts to deal with the update problem are precisely along these lines. Essentially we look for another view (u say), such that between them v and u determine the database uniquely. This is precisely the condition that u is a complement to v. We can then say that for any db with $a = v(db)$ and update $f: a \rightarrow a'$ the translation is such that $u(db)$ remains constant. If no such update is possible, then the original update to the view fails:

{ *translation* }:
let $a = v(db)$, $b = u(db)$ **and** $f: a \rightarrow a'$ be an update
if $\exists db'$ **st** $v(db') = a' \quad u(db') = b$
  define $Tf: db \rightarrow db'$
  otherwise *fail*

That is, the following diagram commutes or the update fails:



Note that this is precisely the same situation as the simple product, as we can regard the valid database states as equivalent to the set of pairs $< v(db), u(db) >$. Alternatively, we could say the product database translation is a special case of this, where we choose for any view v with **range** = $\prod A_{\mu_i}$, the complement u where **range** = $\prod_{\mu \neq \mu_i} A_{\mu}$.

Using complements is not only a nice way of obtaining a translation, it is essential (either implicit or explicit) to achieve reasonable properties. Bancilhon and Spyratos (1981) show that any translation rule T that satisfies:

$\{$ *morphism* $\}$:
$\forall f, f' : A \to A \quad Tf \circ Tf' = T(f \circ f')$
and
$\forall f : A \to A \quad f(a) = a \Rightarrow Tf(db) = db$

where $v(db) = a$, must be given by such a complement. Further translations generated by complements are *process independent*, which enhances their predictability.

There are, of course, many such complement views for a given view. The bigger the view, the more updates it fails. It is therefore sensible to look for a "minimal" view. We can measure minimality using the information ordering, and then ask for a complement that is minimal with respect to this. However, even such a minimal complement cannot be guaranteed to be unique.

The special case of the unconstrained product is obtained if the views are also independent. If a complement is independent, then it is automatically minimal although it is still not unique. However, Cosmadikis and Papadimitriou (1984) show that where the database is relational, there is a unique independent complement so long as we only look at monotonic views. (Here monotonic means that as one adds tuples to the database, tuples are added to the view.)

This special case is not likely to be of general use for complex views, however, so we must be content with having a non-unique complementary view and telling the user what it is. Thus the procedural uncertainty of the user is reduced from the question of "What will this update do?" to "What will stay constant when I do this update?". The latter is static information and more easily conveyed and memorised than the former. It does depend somewhat on the users' preferred learning strategy but at least they have the choice.

9.3.6 Discussion

Clearly, translations generated by complementary views have a lot of advantages in terms of predictability and the consistency of update semantics. The unique complementary independent view derived by Cosmadikis and Papadimitriou is based on assumptions unlikely to obtain for the majority of complex views found in general interface design. The choice of a complementary space becomes a major decision. In particular, it is where task knowledge can be brought into the design process. For instance, if we were trying to form a complimentary view for the variables' index view in the example earlier, we would probably put the procedure names in the complementary view. In fact, this would be an especially interesting choice, as it deliberately makes the complement non-minimal, and is an indication that even where minimal or independent complements can be found, this is not necessarily what is wanted. Of course, this is hardly a novel point, as most operating systems distinguish between the permissions to view and modify data.

Product databases have a lot of advantages:

- All views have an independent complement, implying no failure for internally consistent view updates.
- Reachability, as we can change components independently.
- Procedural uncertainty is minimised, since the complement is obvious.

However, they also have some disadvantages:

- No complex, cross-database views, implying no views like indices, lists of cross-references, etc.
- No hiding, formatting, etc., so all views give complete information on some part of the database.
- No structural change: the product and its views are fixed.

We can alleviate some of these disadvantages. The lack of complex views can be alleviated by having additional *read-only* views that are not necessarily projections. These could be used for navigation but not update. Hiding and formatting can be achieved by allowing several views of each primary view, whose update relative to the primary view is defined either *ad hoc* or using a complementary subview. The semantics of the subview updates can be cross-checked against the explicit view if they are unclear. The issue of structural change is difficult for complementary view translation as well as for product databases, and we consider it in a separate section (§9.5.3).

The real problem with using product databases, is that it is not always easy, or meaningful, to express a design in that way. Having said that, the normalisation process in relational databases attempts to approximate a product database, so it is not an unreasonable goal. Still, for general interfaces we are likely only to approximate the product design in some places, and have to search for more complex complements elsewhere.

For some views it is very difficult even to find a complement. For example, the proper complement for the variables index is not very obvious. If we find ourselves forced to use an *ad hoc* translation strategy, we can still impose a loose complementary view methodology. That is, we can concentrate on what is to be left unchanged in the database when a view is updated, and design our translation around that general idea. It is likely to be helpful for the user to know what is constant, even if this does not uniquely define the update that has occurred. This is an example of searching for a deterministic ground, as discussed in Chapter 6.

In the next section, we consider further the properties of complementary views, in particular their sharing properties, and the predictability of update failure.

9.4 User interface properties and complementary views

We have already noted how the use of a complementary view, where it is possible, increases the predictability of an update strategy, as the user need only know what the complement is and can then infer the result of view updates. In this section we look further at the interface properties of complementary views.

We begin by looking briefly at aliasing problems with complementary views. After this we consider sharing properties and how these relate to sharing of windows as defined in Chapter 4. We then turn our attention to the predictability of update failure, and introduce the concept of a *covering view*. Finally, we assess the security implications of imposing a covering view as an interface requirement.

9.4.1 Agreeing upon the views: aliasing and worse

The designer of a system presents a view to the user. The user sees the contents of the view as the system runs, and infers some idea of what the view is. Of course, as we have seen before, aliasing means that the user and the designer need not agree. Content does not determine identity. Two views may look the same and yet be different. Note that this form of aliasing is more complex than, say, knowing one's position in a document. There we assumed that the user knew the *kind* of view, and the problem was knowing where in the document the view was. Here there may even be confusion about the nature of the view itself.

Imagine we have a programming system that, when the user clicks upon an identifier, gives an index of other references in the program through which she can further navigate. One day, the programmer is looking at a paper listing and wants to get to the line containing the identifier `sum`. Coincidentally the variable `sum` is on the screen, so she clicks this. Unfortunately, it does not reference the line which she wants. Why? Well, the programmer had assumed that the indexing was lexical, to all variables with the given name. In fact, the designer's model was that only the incidents of the same semantic variable were given. The form of the indices for the two views is the same, a list of references, but the views are very different.

Note that this form of aliasing is closely linked to *procedural uncertainty* and *conceptual capture*, as discussed in Chapter 6. This can be attacked only partly through the immediate user interface; the full solution must involve both study of the natural models of users and the production of suitable documentation to explain the nature of the view. A much broader idea of the "interface" is required to address problems of aliasing that arise through procedural uncertainty than those that arise through data uncertainty.

Now, if it is hard for the designer and the user to agree about what they *do* see, how much more difficult it is for them to agree about what they *do not* see. The complementary view is, by its nature, not presented in the interface, and thus misconceptions about what does not change may take a long time to emerge, and the possibility of confusion is enormous. Note that this is not just a problem when the designer explicitly uses a complementary view approach. Whatever view update strategy is chosen, the user is not and *cannot* be aware of all the hidden ramifications of their actions. In the simple case of a product database (if the user is aware of it) the user's actions can be made *local* to the particular attribute. In a more complex setting, the notion of locality depends crucially on viewpoint and the problems are precisely akin to those of determining the complementary view.

There is no easy answer to this issue of agreement between user and designer. The best hope lies in the designer being aware of the problem.

9.4.2 Sharing properties of complementary views

We consider again the sharing properties of windowed systems, where each window is a view of an underlying database. In the section on the algebra of views, we said that if all updates succeed, then view independence guarantees display independence (§9.2.2). Of course, when we consider complementary-view-based translation, some view updates will fail. This means we can be more generous about the views that do not share, as some view updates that would have resulted in display interaction will now fail. Display independence says that the display of one window stays the same as commands are executed in another window. If all the commands are view updates and the complement to the view is constant, then if the first window is a subview of the complement we can guarantee display independence. That is, assuming the complement to the view v is u:

> the window of v' is display independent of the window of v **iff**
> $v' \leq u$

Note that the condition is not only sufficient but necessary.

> **PROOF**:
> If v' is not a subview of u, then we can find two database states db and db' such that:
>
> $$u(db) = u(db') \quad \textbf{and} \quad v'(db) \neq v'(db')$$

However, because v and u between them determine the database we must have:

$$u(db) = u(db') \quad \Rightarrow \quad v(db) \neq v(db')$$

Thus the update:

$$f\colon\ v(\ db\ )\ \rightarrow\ v(\ db'\ )$$

would succeed and would change the value of v'.

Result independence is far more difficult. Consider the following example of two pairs of independent complementary views. We have a database consisting of three, unconstrained integers, (x, y and z). We will consider the two pairs of views:

x with complement (y, z)
y with complement (x, $y \times (\ x + z\ )$)

These views are independent of one another and satisfy the conditions for display independence. We consider the two updates:

$$f_x\ :\ x = 1\ \rightarrow\ x = 2$$
$$f_y\ :\ y = 1\ \rightarrow\ y = 2$$

If we start in the database state ($x = 1$, $y = 1$, $z = 1$), we consider the effects of the translations of these two updates performed in different orders:

{ f_x first then f_y }:
$Tf_x\ :\ (\ 1, 1, 1\ )\ \rightarrow\ (\ 2, 1, 1\ )$
$Tf_y\ :\ (\ 2, 1, 1\ )\ \rightarrow\ (\ 2, 2, -\frac{1}{2}\ )$ $\qquad 1\times(2 + 1)\ =\ 2\times(2 - \frac{1}{2})$

{ f_y first then f_x }:
$Tf_y\ :\ (\ 1, 1, 1\ )\ \rightarrow\ (\ 1, 2, 0\ )$ $\qquad 1\times(1 + 1)\ =\ 2\times(1 + 0)$
$Tf_x\ :\ (\ 1, 2, 0\ )\ \rightarrow\ (\ 2, 2, 0\ )$

That is, the two updates do *not* commute, and hence windows based on these would not be result independent. This is all the more surprising, as our complements were independent and hence "good" ones with no failure. In this example, as the database was a simple product with no constraints, we could have defined the complements so that the updates to x and y did commute (though we might have good task-specific reasons for our definitions). In more complex databases such a choice may not only be inexpedient, but be impossible.

We can give a sufficient (but not necessary) condition for result independence of two independent views v and v' with complements u and u', respectively:

$$\exists\ u_{glb}\quad \textbf{st}\quad u_{glb}\ \leq\ u\quad \textbf{and}\quad u_{glb}\ \leq\ u'$$
$$\textbf{and}\quad u\ \leq\ (\ v \times u_{glb}\ )\quad \textbf{and}\quad u'\ \leq\ (\ v' \times u_{glb}\ )$$

Informally, u_{glb} is complementary to $v \times v'$ and thus all updates are observable from v and v'. However, again, unless one is using a simple policy for finding

complements (such as the product database), such a view may not be possible or desirable.

9.4.3 Predicting failure – covering views

The big advantage of an independent complement is in terms of predictability. If the views are not independent then there are updates that are sometimes legal and sometimes not, depending on the value of the (invisible) complementary view. If we wanted to see all this complementary view in addition to the principal view, then by its definition we would be viewing the *entire* database; exactly what the views of the database are there to avoid. Clearly, we must look for a *covering view* (w) that is small enough to present to the user along with the principal view, and yet has enough information to predict whether a particular update will succeed or fail. We can formulate the property of the covering view as follows:

$$
\begin{array}{l}
\{\ \textit{covering}\ \}: \\
\forall db_1, db_2 \ \ \textbf{st} \ \ (v{\times}w)(db_1) = (v{\times}w)(db_2) \\
\quad \exists db_1' \ \ \textbf{st} \ \ u(db_1') = u(db_1) \\
\qquad\qquad \Rightarrow \ \ \exists db_2' \ \ \textbf{st} \ \ u(db_2') = u(db_2) \\
\qquad\qquad\qquad\qquad\ \ \textbf{and} \ \ v(db_1') = v(db_2')
\end{array}
$$

That is, if for any two database states both the principal view and the covering view are the same, then any consistent update on the first database is consistent for the second. Note again that for any given view and complement, even a minimal covering view is not necessarily unique.

Example

Consider the case of a relational database with the following relations:

employee	
EMP_NAME	EMP_DEPT
Diane	Sales
Fred	Accounts
Tom	Sales
Jane	Clerical
Helen	Sales

department		
DEPT_NAME	BUDGET	other ...
Sales	500	
Clerical	250	
Accounts	3000	
Stores	20	
?Personnel??		

The update view is employee, and every EMP_DEPT must be a valid DEPT_NAME. We try to enter a record for Gillian in the personnel department (just formed); this succeeds or fails depending on whether there is an entry in the department relation for Personnel or not. A suitable covering view for this is

clearly the set of department names.

Another example would be if u were a statement in a Pascal program (and Db the set of valid Pascal programs); then a possible covering view for u might contain a list of all the variable names in the scope of the statement and their types.

In fact, we can ask for a covering view even when the translation is not generated from a complement. In this case we have to amend the definition to say that w is a covering view for an update f to a view v if:

$$\forall db_1, db_2 \quad \textbf{st} \quad (v \times w)(db_1) = (v \times w)(db_2)$$
$$(\exists db_1' \textbf{ st } Tf : db_1 \rightarrow db_1') \Rightarrow (\exists db_2' \textbf{ st } Tf : db_2 \rightarrow db_2')$$

That is information from u and w together is sufficient to tell whether a particular update *to* u will succeed.

NOTE: the covering view deals only with the data uncertainty about the complement's effect on update failure. If users are unaware of what view they are manipulating, or what the constant complement is, then the covering view can only help but may not be sufficient to make this clear. In general, procedural uncertainty is very difficult to handle (viz. 200-page manuals!).

9.4.4 Security

Of course, one of the reasons for restricting access to a view of the database is security. The covering view isn't updatable, but might require information that is not intended to be available to the user.

This is not such a drawback as it seems: the very fact that the updates on the view behave differently depending on the hidden information implies that, in principle, it is possible to infer some of the hidden information. It is precisely this information that is required in the covering view. Thus if one finds that there is some item of information in the covering view that is regarded as confidential, one should think hard about whether that information was indirectly available anyway. This is similar to the case of statistical queries to databases, where one is allowed to ask for summary statistics of the data but is not supposed to have access to the data itself. Depending on the queries allowed it may be possible to infer the raw data from the summary statistics, breaking confidentiality. (Jonge 1983) This argument should not be taken to its extreme: for example, if one enters a random user name and password to a system, then it will behave differently when they are valid and when they are not; however, it would not be reasonable to demand that all the user names and passwords be public! Clearly, one would trade off predictability against security in this case. Even so, predictability analysis and the creation of suitable covering views would actually form a powerful test of system security.

9.5 Dynamic views and structural change

In most of this chapter, we have assumed implicitly that it is meaningful to talk of *a* view, which can be applied to any database state. Typically, views may be more dynamic than that. We can consider three types of dynamism:

- *Content* – The view is defined by some feature of content: for example, the set of all records of people in the Personnel department.
- *Identity* – The view is defined indicatively: for example, *this* collection of records which I want to deal with no matter how their content changes. (The collection may initially be defined by content but is not constrained by it.) Again, in editing text we want to display *this* region of text, whatever changes happen elsewhere.
- *Structure* – Much of the discussion of database views assumes a constant database structure. In general systems this may change, and the set of views available at any time can change accordingly. Many database packages deal very badly with schematic changes to live databases.

The first of these categories is included because it has been described elsewhere as a dynamic view. (Garlan 1986) However, that definition is in fact consistent with the definitions of static view given earlier in this chapter. The complication of it, compared with more syntactic views, is that update semantics are less clear. However, it can be dealt with using complementary views quite adequately. The second category is clearly related to the issue of dynamic pointers. We will reserve the title *dynamic view* for this category of view. The third category of structural change is very hard to deal with and we will see by example how easy it is to make a mess of this.

9.5.1 Dynamic views

We look at general views where we want the semantic identity of the view to be preserved. As with pointer spaces we have two distinguishing factors, the set of views available changes with the object viewed, and the notion of "sameness" between views depends on the operations that caused the database change.

The first of these can be captured by using a set of *valid views* for each database state, in a similar fashion to the valid pointers to an object we considered in the previous chapter. For the "sameness" property we can use the idea of a *pull* function again. For each update F, we associate a pull function $F.pull$ which takes views to their "natural" equivalents:

$$\begin{array}{l} \forall\ db \in Db,\ v \in valid\_views(\ db\ ) \\ \quad \textbf{let}\quad db', pull\ =\ F(\ db\ ) \\ \quad \textbf{then}\ \ pull(\ v\ ) \in valid\_views(\ db'\ ) \end{array}$$

When we discussed various update strategies, we regarded *process independence* as being a useful trait. That is, the form of the update was unimportant, only the initial and final states mattered. This will almost always *not* be the case for dynamic views. We may easily be able to perform two different series of updates which would yield the same final database state, but affect the dynamic views in a totally different manner. We must not only know *what* an update does, but *how* it does it.

As an example of the importance of this, consider two records in a database:

$< TOM, 1973 >$
$< FRED, 1971 >$

We start with a view looking at Tom's record. We then edit the database exchanging first the names, and then the dates. A view defined by content would still look at $< TOM, 1973 >$, but a view based on identity would be of the record that now says $< FRED, 1971 >$.

The problem of view update translation is now correspondingly more complex: given $a = u(db)$ and $f: a \rightarrow a'$ we want a translation $Tf: db \rightarrow db'$ such that $(Tf\,.\,pull(u))\,(db') = a'$.

It is similarly more complex to talk about complements for such views. We cannot talk about *a* complement of a specific view, but instead of a complement *function* from views to views. This complement function must of course be preserved by the *pull* functions. If we let the set of complement views be $V_c : Db \rightarrow B$, a possibly different set of views than V, we get the following condition:

$comp: \quad V \rightarrow V_c$
$\forall v \in V, F \in U, db \in Db$
$\quad comp(\ F\,.\,pull(\ db, v\ )\ ) \ = \ F\,.\,pull(\ db, comp(\ v\ )\ )$

For the map to define a complementariness relation, we need the view and its complement to determine the database state:

$\forall db, db' \in Db, v \in valid\_views(\ db\ ), v' \in valid\_views(\ db'\ )$
let $u = comp(\ v\ ), \quad u' = comp(\ v'\ )$
then
$v(\ db\ ) = v'(\ db'\ )$ **and** $u(\ db\ ) = u'(\ db'\ ) \quad \Rightarrow \quad db = db'$

That is, if we have a value for the view, and a value for the complement, there is only one possible set of database state and view with these as their values. We can then use this to define a translation scheme for update, and the new view to use. It does not give us the general pull function for other views. This may not matter if there is only one view of interest, but if this is not so we need more structure again to obtain the relevant function. Alternatively, we can use the

existing knowledge of pointer spaces and model dynamic views using these.

9.5.2 Using pointer spaces to model dynamic views

The properties one wants of dynamic views are clearly similar to those of dynamic pointers, so it is natural to consider using pointer spaces and subobject projections to give dynamic views. The notions of determinacy and independence generated by subobject projections are consistent with the definition of information ordering and independence for views. The only difference is that the subobject projection, as a function of a block b, is defined only for objects with a given set of valid pointers. This is precisely what we expect for dynamic views. Again, even when a sequence of operations maps back to an object with the same valid pointers, there is no guarantee that statically identical blocks are semantically the same (i.e. pulled to each other). So, for instance, we might operate on the string "abcde" by *delete*(3, _) followed by *insert*(2, "*c*", _):

$$abcde \xrightarrow{delete(\,3\,)} abde \xrightarrow{insert(\,2,\,c)} abcde$$

However, $pull(\,(\,2, 3\,)\,) = (\,3, 3\,) \neq (\,2, 3\,)$. As block pointers behave so much like the intended behaviour of dynamic views, it is natural to identify the object component of their subobject projection with the view.

This approach is taken further in Dix (1987b) and demonstrates the usefulness of the dynamic pointer approach. However, the use of dynamic pointers cannot hide the fundamental complexity of dynamic views. When we considered static views, we saw how difficult it can be even for the designer and the user to retain a common understanding both of what they do see and of what they do not see. This is even more difficult and requires more care on the part of the designer when the views are dynamic and essentially "move about" the database.

9.5.3 Structural change

Some years ago a colleague was demonstrating a powerful (and expensive) commercial relational database that he was evaluating. After seeing some of the basic operations, I asked how well it coped with changes to the database schema. Within seconds he added a new field to an existing relation. I was duly impressed. However, I noticed that the database manager had filled the empty fields with a special **NULL** entry. One of the options for a field was that **NULL**s were not allowed. What would happen if you asked it to add a new non-**NULL** field. He tried it and, sensibly enough, the database refused. Presumably you obtained a non-**NULL** field by first creating a normal field, filling it with non-

NULL entries and then changing its attributes. What would happen if you made a mistake? One of the fields in the relation we were looking at had some **NULL** and some filled entries. Try changing that field to be non-**NULL**, I suggested. He did and the database complied. All the records with **NULL**s in them were removed!

Structural change is hard. The designers of the above database had obviously thought about it but had not considered all of the consequences. Further, error recovery mechanisms may not be as good when updates involve structural change.

A computer installation I once worked in used a large mainframe with the manufacturer's standard operating system. One of the attributes of a disk file was the number of tape backups that would be made automatically when the file was updated. Apparently this afforded a high degree of data security. However, one day an old version of a large file was required, the tape was found and an attempt made to restore it. Unfortunately it failed. In order to restore a file it had to return it to the same device with the same attributes, such as block size and file type, as when the backup was made. These attributes had changed and the values when the backup was made had been forgotten. The number of possible attribute combinations was large, and the information was not explicitly available from the backup tape itself. The relevant data were recreated from paper records!

The recovery mechanism above was designed to work only in the structurally static case. If the operating system's designers had thought about the possibility of structural change, they could easily have added some standard tape header giving the necessary attribute and device information.

If we have a structural view of some underlying data, the update semantics are clearly quite complex. The notion of a complementary view does not help us that much: the things we do not see may not change, they may simply cease to exist! A small part of the view, say a file icon, represents a vast and valuable piece of data. Small changes to the view may mean enormous changes to the data. In simple file manipulation systems this is recognised and often special dialogues are initiated if the effect of an action is gross. Most computer filing systems are configured as, at most, a simple hierarchy. With these, it is not too unreasonable to expect users to understand the impact of their actions and to expect systems to detect potential troublespots. If the organisation of data becomes more complex, say with an object-oriented database or a hypermedia system, it becomes harder for the user to appreciate the potential scope of their actions and harder for the system to tell which actions it should warn about and which it should not. For instance, in the Unix file system a single file may have several "links", that is, it may have several access paths through the otherwise hierarchical file system structure. Removing any link but the last merely removes an access path: the file is still there and (if you can find it) can be recovered. Removing the last link destroys the file, yet there is no readily

apparent difference between these two operations. The consistency is very useful at a systems programming level, but is potentially disastrous at the level of the user interface. As with simpler systems, if the designer is too zealous in producing dialogue boxes and confirmation messages the user will, of course, reply habitually, and the benefit is lost. Finding the appropriate balance is not easy, and there appear to be no easy formulae.

The trouble with updating structural views is that they are powerful. On the one hand, this power can be of great *value*, allowing the appropriate structuring of information and adding to its usefulness thereby. For instance, the database described above allowed (albeit disastrously in some cases) structural change. If this were not supported, a costly copying process between the old and new formats would be needed. Of course, with that power comes also *risk*. Inevitably, powerful actions will occasionally go wrong, and this emphasises the need for equally powerful recovery systems. Recovery systems that work only when the data structure is static and assume that structural change will be performed only by faultless "wizards" are not worth a lot. The Macintosh wastebin is an example of a recovery mechanism for a structural view, so it is not impossible (however, even this tends to "empty itself" in certain circumstances).

We come back to the fact that structural change is hard. This means that we have to think more about it. So many potential designs are considered only in structurally static scenarios; this may well represent a large proportion of the use, but side-steps some of the most difficult design issues, relegating them to a phase late in the production process when they may be fudged.

9.6 Conclusions

We have investigated the problem of updating an object through views to it. Although *ad hoc* methods have some advantages, it is necessary to use some more predictable method to reduce procedural uncertainty. If the database can be seen as a product then the problems of update predictability and reachability are easy. Further, we can easily test for sharing and interference between updates. Traditional database theory tried as far as possible to simulate this position by the use of normalisation rules, and in a sense follows the route (strongly favoured in programming language design too) of pushing semantics into syntax. This often means trading power for predictability. This might be a good goal to aim for, but is unlikely to hold for many more complex views.

We have seen that researchers into database views have used the concept of a complementary view, which stays the same whilst the principal view is updated, as a way of defining the update for the database. This is more likely to succeed as a strategy than searching for a product formulation, and is less restrictive, yet it retains some of the predictability properties. In particular, it is only necessary

to know what the complementary view is, to know the effect of updates.

Unfortunately, the failure semantics of updates are not in general predictable for complementary view updates and thus it is necessary to introduce the idea of a *covering view*. This contains sufficient information from the complement to predict which updates to the principal view will fail. The covering view would not be updatable in itself, and perhaps would have weaker sharing restrictions.

We saw that both the primary view and its complement have *aliasing* problems. Agreeing on the view presented to the user has some difficulties, but at least there is a common reference and differences in interpretation are likely to become apparent. The complementary view, by its very nature, is unseen and thus agreement is far more difficult. In both cases the aliasing is a form of *procedural uncertainty* and thus difficult to address purely within the on-line system. Solutions are bound to encompass the whole analysis of user's models and the production of documentation and training.

Although DP databases tend to have fixed structures, and the views on them tend to be gross and static, the same is not true of more general databases (e.g. graphics, program syntax trees) and thus we need to look at the case of dynamic views and structural change. Both these problems are considerably more difficult than the static case, and even that is not straightforward.

In conclusion, it is wise to search as far as possible for a product formulation of one's problem space and, failing that, look for complementary views. Even where these fail and an *ad hoc* approach is necessary, it will probably be advantageous to take a complementary-view-flavoured approach, and concentrate on what is to be left unchanged by an update. There is little reason for not including covering views, unless the noise these cause far outweighs the annoyance of unpredictable failure, or if it involves an extreme and pedantic case such as that of passwords and user names. For two rules to sum up the chapter:

- Make sure the users know *what* they see.
- Make sure the users know what they do *not* see.