To see how these rules work, imagine that the user has just seen the typing mistake and thus the contents of working memory (w.m.) are

```
(GOAL perform unit task)
(TEXT task is insert space)
(TEXT task is at 5 23)
(CURSOR 8 7)
```

`TEXT` refers to the text of the manuscript that is being edited and `CURSOR` refers to the insertion cursor on the screen. Of course, these items are not actually located in working memory – they are external to the user – but we assume that knowledge from observing them is stored in the user's working memory.

The location (5,23) is the line and column of the typing mistake where the space is required. However, the current cursor position is at line 8 and column 7. This is of course acquired into the user's working memory by looking at the screen. Looking at the four rules above (`SELECT-INSERT-SPACE`, `INSERT-SPACE-DONE`, `INSERT-SPACE-1` and `INSERT-SPACE-2`), only the first can fire. The condition for `SELECT-INSERT-SPACE` is:

```
(AND (TEST-GOAL perform unit task)
          true because (GOAL perform unit task) is in w.m.
     (TEST-TEXT task is insert space)
          true because (TEXT task is insert space) is in w.m.
     (NOT (TEST-GOAL insert space))
          true because (GOAL insert space) is not in w.m.
     (NOT (TEST-NOTE executing insert space)) )
          true because (NOTE executing insert space) is not in w.m.
```

So, the rule fires and its action is performed. This action has no external effect in terms of keystrokes, but adds extra information to working memory. The `(LOOK-TEXT task is at %LINE %COL)` looks for a corresponding entry and *binds* `LINE` and `COL` to 5 and 23 respectively. These are variables, somewhat as in a normal programming language, which are referred to again in other rules.

The contents of working memory after the firing of rule `SELECT-INSERT-SPACE` are as follows (note that the order of elements of working memory is arbitrary):

```
(GOAL perform unit task)
(TEXT task is insert space)
(TEXT task is at 5 23)
(NOTE executing insert space)
(GOAL insert space)
(LINE 5)
(COL 23)
(CURSOR 8 7)
```

At this point neither rule `SELECT-INSERT-SPACE` nor `INSERT-SPACE-DONE` will fire as the entry `(GOAL insert space)` will make their conditions false. As `LINE` is bound to 5 and `COL` is bound to 23, the condition `(TEST-CURSOR %LINE %COL)` will be false also, and hence only rule `INSERT-SPACE-1` can fire.

After this rule's actions have been performed, the working memory will include the entry (GOAL move cursor to 5 23). The rules for moving the cursor are not included here, but would be quite extensive, moving up/down and right/left depending on the relative positions of the cursor and the target location. Eventually, assuming the cursor movement is successful, the cursor would be at (5,23) whence rule INSERT-SPACE-2 would be able to fire. This would perform the keystrokes: 'I', SPACE and ESC, which in **vi** puts the editor into insert mode, types the space and then leaves insert mode. The action also removes the 'insert space' goal from working memory as this goal has been achieved.

Now the goal has been removed, the second rule INSERT-SPACE-DONE is free to fire, which 'tidies up' working memory. In particular, it 'unbinds' the variables LINE and COL, that is it removes the bindings for them from working memory.

Notice that the rules did not fire in the order they were written. Although they look somewhat like the if–then–else commands one would get in a standard programming language, they behave very differently. The rules are all active and at each moment any rule that has its conditions true may fire. Some rules may never fire; for instance, if the cursor is at the correct position the third rule would not fire. Furthermore, the same rule may fire repeatedly; for example, if we were to write out the production rules for moving the cursor, one rule may well be

```
(MOVE-UP
IF  (AND  (TEST-GOAL move-up)
          (TEST-CURSOR-BELOW %LINE) )
THEN (    (DO-KEYSTROKE 'K') ))
```

This rule is to type 'K' (the **vi** command to move the cursor up one line) while the cursor is below the desired line. It will, of course, be constantly refired until the cursor is at the correct line.

Notice that the keystrokes for actually inserting the space, once you are at the right position, have been *proceduralized.* That is, the user does not go through the subgoals 'enter insert mode', 'type space', 'leave insert mode'. For a complex insertion, it is quite likely that the user will perform exactly these goals. However, the act of inserting a single space is assumed to be so well rehearsed that it is stored as a single chunk. That is, the rules above represent *expert* knowledge of the **vi** editor.

Of course, novices may well do exactly the same keystrokes as the experts, but the way they store the knowledge will be different. To cope with this CCT has a set of 'style' rules for novices. These limit the form of the conditions and actions in the production rules. Basically, novices are expected to test constantly all the rules in their working memory and to check for feedback from the system after every keystroke. Thus a set of 'novice' rules would not include the proceduralized form of insert space. Bovair, Kieras and Polson provide a list of many style rules which can be used to embody certain psychological assumptions about the user (novice/expert distinction is only one) in a CCT description [27].

The rules in CCT need not represent error-free performance. They can be used to explain error phenomena, though they cannot predict them. For instance, the rules above for inserting a space are 'buggy' – they do not check the editor's mode. Imagine you had just been typing the 'cognitive' in 'cognitivecomplexity theory' (with the space missing), you think for a few minutes and then look again at the screen and notice that the space is missing. The cursor is at the correct position for the space, so rule INSERT-SPACE-1 never gets fired and we go directly through the sequence: SELECT-INSERT-SPACE, INSERT-SPACE-2 then INSERT-SPACE-DONE. You type 'i', a space and then escape. However, the 'i' assumes that you are in **vi**'s command mode, and is the command to move the editor into insert mode. If, however, after typing 'cognitive' you had not typed escape, to get you back into command mode, the whole sequence would be done in insert mode. The text would read: 'cognitiveI complexity theory'.

The CCT rules are closely related to GOMS-like goal hierarchies; the rules may be generated from such a hierarchy, or alternatively, we may analyze the production rules to obtain the goal tree:

```
GOAL: insert space
.      GOAL: move cursor - if not at right position
.      PRESS-KEY-I
.      PRESS-SPACE
.      PRESS-ESCAPE
```

The stacking depth of this goal hierarchy (as described for GOMS) is directly related to the number of (GOAL ...) terms in working memory.

In fact, the CCT rules can represent more complex plans than the simple sequential hierarchies of GOMS. The continuous activity of all production rules makes it possible to represent concurrent plans. For example, one could have one set of production rules representing the goal of writing a book, and another set representing the goal of drinking tea. These rules could both be active simultaneously, thus allowing an author to drink tea whilst typing. Despite this apparent flexibility, CCT is not normally used in this way. It is not clear why this is, except that CCT, like GOMS, is aimed at low-level, *proceduralized* goals – that is, the *unit task*. It is reasonable that successive unit tasks be chosen from different activities: the author may delete a word, have a drink, do a word search, but each time a complete unit task would be performed – the author does not take a drink of tea in the middle of deleting a word.

We have seen how CCT rules may be informally analyzed to discuss issues of proceduralization and error behaviour, and how we can relate them to GOMS-like goal hierarchies. However, the main aim of CCT is (as its name suggests) to be able to measure the complexity of an interface.

Basically, the more production rules in the CCT description the more difficult the interface is to learn. This claim rests on the assumption that the production rules represent reasonably accurately the way knowledge is stored and therefore that the time taken to learn an interface is roughly proportional to the number of rules you have to learn.

We have only discussed the user side of CCT here. If the cognitive user description is complemented by a description of the system, it is claimed that one can predict the difficulty of the mapping between the user's goals and the system model. The generalized transition networks which describe the system grammar themselves have a hierarchical structure. Thus both the description of the user and that of the system can be represented as hierarchies. These can then be compared to find mismatches and to produce a measure of dissonance.

There are various problems with CCT. As with many 'rich' description methods, the size of description for even a part of an interface can be enormous. Furthermore, there may be several ways of representing the same user behaviour and interface behaviour, yielding different measures of dissonance. To some extent this is catered for by the novice style rules, but there is no such set of rules for the system description.

Another problem is the particular choice of notations. Production rules are often suggested as a good model of the way people remember procedural knowledge, but there are obvious 'cludges' in the CCT description given above. In particular, the working memory entry `(NOTE executing insert space)` is there purely to allow the `INSERT-SPACE-DONE` rule to fire at the appropriate time. It is not at all clear that it has any real cognitive significance. One may also question whether the particular notation chosen for the system is critical to the method. One might choose to represent the system using any one of the dialog description notations in Chapter 8. Different notations would probably yield slightly different measures of dissonance.

However, one should regard CCT as an engineering tool giving one a rough measure of learnability and difficulty combined with a detailed description of user behaviour. This can then be used by analysts employing their professional expertise. Arguably, the strength of the central idea of CCT lies beyond the particular notations used.

6.7.3 Problems and extensions of goal hierarchies

The formation of a goal hierarchy is largely a *post hoc* technique and runs a very real risk of being defined by the computer dialog rather than the user. One way to rectify this is to produce a goal structure based on pre-existing manual procedures and thus obtain a natural hierarchy [130]. To be fair, GOMS defines its domain to be that of expert use, and thus the goal structures which are important are those which users develop out of their use of the system. However, such a natural hierarchy may be particularly useful as part of a CCT analysis, representing a very early state of knowledge.

On the positive side, the conceptual framework of goal hierarchies and user goal stacks can be used to express interface issues, not directly addressed by the notations above. For instance, early automated teller machines gave the customers the money before returning their cards. Unfortunately, this led to many customers leaving their cards behind. This was despite on-screen messages telling them to wait. This is referred to as a problem of *closure*. The user's principal goal is to get money; when that goal is satisfied, the user does not complete or close the various subtasks which still remain open:

```
GOAL: GET-MONEY
.   GOAL: USE-ATM
.   .    INSERT-CARD
.   .    ENTER-PIN
.   .    ENTER-AMOUNT
.   .    COLLECT-MONEY
            << outer goal now satisfied goal stack popped >>
.   .    COLLECT-CARD   -   subgoal operators missed
```

Banks (at least some of them) soon changed the dialog order so that the card was always retrieved before the money was dispensed. A general rule that can be applied to any goal hierarchy from this is that no higher level goal should be satisfied until all subgoals have been satisfied. However, it is not always easy to predict when the user will consider a goal to have been satisfied. For instance, one of the authors has been known to collect his card and forget the money!

6.8 Linguistic models

The user's interaction with a computer is often viewed in terms of a language, so it is not surprising that several modelling formalisms have developed centred around this concept. Several of the dialog notations described in Chapter 8 are also based on linguistic ideas. Indeed, BNF grammars are frequently used to specify dialogs. The models here, although similar in form to dialog design notations, have been proposed with the intention of understanding the user's behaviour and analyzing the cognitive difficulty of the interface.

6.8.1 BNF

Representative of the *linguistic approach* is Reisner's use of Backus–Naur Form (*BNF*) rules to describe the dialog grammar [206]. This views the dialog at a purely syntactic level, ignoring the semantics of the language. BNF has been used widely to specify the syntax of computer programming languages, and many system dialogs can be described easily using BNF rules. For example, imagine a graphics system which has a line-drawing function. To select the function the user must select the 'line' menu option. The line-drawing function allows the user to draw a polyline, that is a sequence of line arcs between points. The user selects the points by clicking the mouse button in the drawing area. The user double clicks to indicate the last point of the polyline.

```
draw-line      ::=  select-line + choose-points
                                + last-point
select-line    ::=  position-mouse + CLICK-MOUSE
choose-points  ::=  choose-one
                 |  choose-one + choose-points
choose-one     ::=  position-mouse + CLICK-MOUSE
last-point     ::=  position-mouse + DOUBLE-CLICK-MOUSE
position-mouse ::=  empty | MOVE-MOUSE + position-mouse
```

The names in the description are of two types: *non-terminals*, shown in lower case, and *terminals*, shown in upper case. Terminals represent the lowest level of user behaviour, such as pressing a key, clicking a mouse button or moving the mouse. Non-terminals are higher-level abstractions. The non-terminals are defined in terms of other non-terminals and terminals by a definition of the form

```
name       ::= expression
```

The '::=' symbol is read as 'is defined as'. Only non-terminals may appear on the left of a definition. The right hand side is built up using two operators '+' (sequence) and '|' (choice). For example, the first rule says that the non-terminal `draw-line` is defined to be `select-line` followed by `choose-points` followed by `last-point`. All of these are non-terminals, that is they do not tell us what the basic user actions are. The second rule says that `select-line` is defined to be `position-mouse` (intended to be over the 'line' menu entry) followed by `MOUSE-CLICK`. This is our first terminal and represents the actual clicking of a mouse.

To see what `position-mouse` is, we look at the last rule. This tells us that there are two possibilities for `position-mouse` (separated by the '|' symbol). One option is that `position-mouse` is `empty` – a special symbol representing no action. That is, one option is not to move the mouse at all. The other option is to do a `MOVE-MOUSE` action followed by `position-mouse`. This rule is recursive, and this second `position-mouse` may itself either be empty or be a `MOVE-MOUSE` action followed by `position-mouse`, and so on. That is, `position-mouse` may be any number of `MOVE-MOUSE` actions whatsoever.

Similarly, `choose-points` is defined recursively, but this time it does not have the option of being empty. It may be *one or more* of the non-terminal `choose-one` which is itself defined to be (like `select-line`) `position-mouse` followed by `MOUSE-CLICK`.

The BNF description of an interface can be analyzed in various ways. One measure is to count the number of rules. The more rules an interface requires to use it, the more complicated it is. This measure is rather sensitive to the exact way the interface is described. For example, we could have replaced the rules for `choose-points` and `choose-one` with the single definition

```
choose-points  ::= position-mouse + CLICK-MOUSE
        | position-mouse + CLICK-MOUSE + choose-points
```

A more robust measure also counts the number of '+' and '|' operators. This would, in effect, penalize the more complex single rule. Another problem arises with the rule for `select-line`. This is identical to the `choose-one` rule. However, the acts of selecting a menu option and choosing a point on a drawing surface are obviously so different that they must surely be separated. Decisions like this about the structure of a BNF description are less of a problem in practice than the corresponding problems we had with CCT.

In addition to these complexity measures for the language as a whole, we can use the BNF definition to work out how many basic actions are required for a particular task, and thus obtain a crude estimate of the difficulty of that task.

The BNF description above only represented the user's actions, not the user's perception of the system's responses. This input bias is surprisingly common amongst cognitive models, as we will discuss in Section 6.9. Reisner has developed extensions to the basic BNF descriptions which attempt to deal with this by adding 'information-seeking actions' to the grammar.

6.8.2 Task–action grammar

Measures based upon BNF have been criticized as not 'cognitive' enough. They ignore the advantages of consistency both in the language's structure and in its use of command names and letters. *Task–action grammar* (*TAG*) [193] attempts to deal with some of these problems by including elements such as parametrized grammar rules to emphasize consistency and encoding the user's world knowledge (for example, up is the opposite of down).

To illustrate consistency, we consider the three UNIX commands: cp (for copying files), mv (for moving files) and ln (for linking files). Each of these has two possible forms. They either have two arguments, a source and destination filename, or have any number of source filenames followed by a destination directory:

```
copy  ::=   'cp' + filename + filename
          | 'cp' + filenames + directory
move  ::=   'mv' + filename + filename
          | 'mv' + filenames + directory
link  ::=   'ln' + filename + filename
          | 'ln' + filenames + directory
```

Measures based upon BNF could not distinguish between these consistent commands and an inconsistent alternative – say if ln took its directory argument first. Task–action grammar was designed to reveal just this sort of consistency. Its description of the UNIX commands would be

```
file-op[Op]          :=   command[Op] + filename + filename
                        | command[Op] + filenames + directory
command[Op=copy]  :=   'cp'
command[Op=move]  :=   'mv'
command[Op=link]  :=   'ln'
```

This captures the consistency of the commands and closely resembles the original textual description. One would imagine that a measure of the complexity of the language based on the TAG description would be better at predicting actual learning and performance than a simple BNF one.

As well as handling consistency well, TAG has features for talking about 'world knowledge'. For example, imagine we have two command line interfaces for moving a mechanical turtle around the floor.

Command interface 1

```
movement[Direction]
              :=  command[Direction] + distance + RETURN
command[Direction=forward]    := 'go 395'
command[Direction=backward]   := 'go 013'
command[Direction=left]       := 'go 712'
command[Direction=right]      := 'go 956'
```

Command interface 2

```
movement[Direction]
              :=  command[Direction] + distance + RETURN
command[Direction=forward]    := 'FORWARD'
command[Direction=backward]   := 'BACKWARD'
command[Direction=left]       := 'LEFT'
command[Direction=right]      := 'RIGHT'
```

The first interface may not be as silly as it seems; the command 'go 395' could refer to the address of a machine-code routine which performs the appropriate movement. However, it is absolutely clear that the second interface is preferable to the first. TAG includes a special form `known-item` which is used to denote information that the user will already know, and thus not need to learn in order to use the system. Using this form, the TAG rules for the second interface are rewritten

Command interface 2

```
movement[Direction]
                 :=   command[Direction] + distance + RETURN
command[Direction]:=   known-item[Type=word,Direction]
*   command[Direction=forward]   := 'FORWARD'
*   command[Direction=backward]  := 'BACKWARD'
*   command[Direction=left]      := 'LEFT'
*   command[Direction=right]     := 'RIGHT'
```

The starred rules can be generated from the second rule using the user's world knowledge. They are included in any TAG description for completeness, but are not counted in any measure of complexity.

Sometimes it may not be clear what the appropriate command is, but once we know one, the rest become obvious. For example, consider a simple database displaying a list of records. We are expecting two commands, one to move up the list to the previous record, and another to move down the list to the next record. There are several options for the commands, for instance UP/DOWN, PREVIOUS/NEXT, possibly in upper or lower case, possibly also just the first letter of the relevant word. In addition, one might have mixed-up command sets such as UP/NEXT or N/previous. The fact that any of the former set of commands is easier to learn than the mixed-up commands is called *congruence*. TAG has a notation to describe the congruence of an interface. The notation `F('next')` is used to denote the feature set related to the word 'next'. That is, next/previous. With this notation a congruent grammar requires only one 'real' rule, such as

```
browse[Direction]  :=  F('next') + return
*  browse[Direction=up]    :=  'previous' + return
*  browse[Direction=down]  :=  'next' + return
```

We have seen that the notation allows one to say that the commands RIGHT and LEFT are consistent for opposite actions. How do we know that the user regards the opposite of RIGHT to be LEFT rather than WRONG? Obviously, the inclusion of world knowledge depends upon the user of the system – the above certainly assumes that the user's language is English. The designer is obviously responsible for inputting this world knowledge into the TAG description and its validity will depend on the professional judgement of the designer. However, TAG will make these assumptions clear and thus, by highlighting them, hold them up for inspection.

6.9 The challenge of display-based systems

Both goal hierarchical and grammar-based techniques were initially developed when most interactive systems were command line or at most keyboard and cursor based. There are significant worries about how well these approaches can therefore generalize to deal with more modern windowed and mouse-driven interfaces.

Both families of techniques largely ignore system output – what the user sees. The implicit assumption is that the users know exactly what they want to do and execute the appropriate command sequences blindly. There are exceptions to this. We have already mentioned how Reisner's BNF has been extended to include assertions about output. In addition, TAG has been extended to include information about how the display can affect the grammar rules.

Another problem for grammars is the lowest-level lexical structure. Pressing a cursor key is a reasonable *lexeme*, but moving a mouse one pixel is less sensible. In addition, pointer-based dialogs are more display oriented. Clicking a cursor at a particular point on the screen has a meaning dependent on the current screen contents. This problem can be partially resolved by regarding operations such as 'select region of text' or 'click on quit button' as the terminals of the grammar. If this approach is taken, the detailed mouse movements and parsing of mouse events in the context of display information (menus etc.) are abstracted away.

Goal hierarchy methods have different problems, as more display-oriented systems encourage less structured methods for goal achievement. Instead of having well-defined plans, the user is seen as performing a more exploratory task, recognizing fruitful directions and backing out of others. Typically, even when this exploratory style is used at one level, we can see within it and around it more goal-oriented methods. So, for example, we might consider the high-level goal structure

```
WRITE_LETTER
.    FIND_SIMILAR_LETTER
.    COPY_IT
.    EDIT_COPY
```

However, the task of finding a similar letter would be exploratory, searching through folders, etc. Such recognition-based searching is extremely difficult to represent as a goal structure. Similarly the actual editing would depend very much on non-planned activities: 'ah yes, I want to reuse that bit, but I'll have to change that'. If we then drop to a lower level again, goal hierarchies become more applicable. For instance, during the editing stage we might have the 'delete a word' subdialog:

```
DELETE_WORD
.    SELECT_WORD
.    .    MOVE_MOUSE_TO_WORD_START
.    .    DEPRESS_MOUSE_BUTTON
.    .    MOVE_MOUSE_TO_WORD_END
.    .    RELEASE_MOUSE_BUTTON
.    CLICK_ON_DELETE
.    .    MOVE_MOUSE_TO_DELETE_ICON
.    .    CLICK_MOUSE_BUTTON
```

Thus goal hierarchies can partially cope with display-oriented systems by an appropriate choice of level, but the problems do emphasize the rather prescriptive nature of the cognitive models underlying them.

These problems have been one of the factors behind the growing popularity of *situated action* [230] and *distributed cognition* [135, 119] in HCI (see also Chapter 14). Both approaches emphasize the way in which actions are contingent upon events and determined by context, rather than being preplanned. At one extreme, protagonists of these approaches seem to deny any planned actions or long-term goals. On the other side, traditional cognitive modellers are modelling *display-based cognition* using production rules and similar methods, which include sensory data within the models.

At a low level, chunked expert behaviour is modelled effectively using hierarchical or linguistic models, and is where the *keystroke-level model* (discussed later in this chapter) has proved effective. In contrast, it is clear that no amount of cognitive modelling can capture the activity during the writing of a poem. Between these two, cognitive models will have differing levels of success and utility. Certainly models at all but the lowest levels must take into account the user's reactions to feedback from the system, otherwise they cannnot address the fundamental issue of *interactivity* at all.

6.10 Physical and device models

6.10.1 Keystroke-level model

Compared with the deep cognitive understanding required to describe problem-solving activities, the human motor system is well understood. KLM (Keystroke-Level Model [36]) uses this understanding as a basis for detailed predictions about user performance. It is aimed at unit tasks within interaction – the execution of simple command sequences, typically taking no more than 20 seconds. Examples of this would be using a search and replace feature, or changing the font of a word. It does not extend to complex actions such as producing a diagram. The assumption is that these more complex tasks would be split into subtasks (as in GOMS) before the user attempts to map these into physical actions. The task is split into two phases:

acquisition of the task, when the user builds a mental representation of the task;

execution of the task using the system's facilities.

KLM only gives predictions for the latter stage of activity. During the acquisition phase the user will have decided how to accomplish the task using the primitives of the system, and thus, during the execution phase, there is no high-level mental activity – the user is effectively expert. KLM is related to the GOMS model, and can be thought of as a very low-level GOMS model where the method is given.

The model decomposes the execution phase into five different physical motor operators, a mental operator and a system response operator:

K Keystroking, actually striking keys, including shifts and other modifier keys.

B Pressing a mouse button.

P Pointing, moving the mouse (or similar device) at a target.

H Homing, switching the hand between mouse and keyboard.

D Drawing lines using the mouse.

M Mentally preparing for a physical action.

R System response which may be ignored if the user does not have to wait for it, as in copy typing.

The execution of a task will involve interleaved occurrences of the various operators. For instance, imagine we are using a mouse-based editor. If we notice a single character error we will point at the error, delete the character and retype it, and then return to our previous typing point. This is decomposed as follows:

1	move hand to mouse	**H**[mouse]
2	position mouse after bad character	**PB**[LEFT]
3	return to keyboard	**H**[keyboard]
4	delete character	**MK**[DELETE]
5	type correction	**K**[char]
6	reposition insertion point	**H**[mouse]**MPB**[LEFT]

Notice that some operators have descriptions added to them, representing which device the hand homes to (for example, [mouse]) and what keys are hit (for example, LEFT – the left mouse button).

The model predicts the total time taken during the execution phase by adding the component times for each of the above activities. For example, if the time taken for one keystroke is t_K, then the total time doing keystrokes is

$$T_K = 2t_K$$

Similar calculations for the rest of the operators give a total time of

$$\begin{aligned} T_{\text{execute}} &= T_K + T_B + T_P + T_H + T_D + T_M + T_R \\ &= 2t_K + 2t_B + t_P + 3t_H + 0 + 2t_M + 0 \end{aligned}$$

In this example, the system response time was zero. However, if the user had to wait for the system then the appropriate time would be added. In many typing tasks, the user can type ahead anyway and thus there is no need to add response times. Where needed, the response time can be measured by observing the system.

The times for the other operators are obtained from empirical data. The keying time obviously depends on the typing skill of the user and different times are thus used for different users. Pressing a mouse button is usually quicker than typing (especially for two-finger typists), and a more accurate time prediction can be made by separating out the button presses **B** from the rest of the keystrokes **K**. The pointing

Table 6.1 Times for various operators in the KLM (adapted from Card, Moran and Newell [37])

Operator	Remarks	Time (s)
K	Press key	
	good typist (90 wpm)	0.12
	poor typist (40 wpm)	0.28
	non-typist	1.20
B	Mouse button press	
	down or up	0.10
	click	0.20
P	Point with mouse	
	Fitts' law	$0.1 \log_2 (D/S + 0.5)$
	average movement	1.10
H	Home hands to and from keyboard	0.40
D	Drawing – domain dependent	–
M	Mentally prepare	1.35
R	Response from system – measure	–

time can be calculated using Fitts' law (see Chapter 1), and thus depends on the size and position of the target[1]. Alternatively, a fixed time based on average within screen pointing can be used. Drawing time depends on the number and length of the lines drawn, and is fairly domain specific, but one can easily use empirical data for more general drawing tasks. Finally, homing time and mental preparation time are assumed constant. Typical times are summarized in Table 6.1.

The mental operator is probably the most complex part of KLM. Remember that the user is assumed to have decided what to do, and how to do it. The mental preparation is thus just the slight pauses made as the user recalls what to do next. There are complicated heuristics for deciding where to put **M** operators, but they all boil down to the level of chunking (see Chapter 1 for a discussion of chunking). If the user types a word, or a well-known command name, this will be one chunk, and hence only require one mental operator. However, if a command name were an acronym which the user was recalling letter by letter, then we would place one **M** operator per letter.

The physical operator times all depend on the skills of the user. Also the mental operator depends on the level of chunking, and hence the expertise of the user. You must therefore decide before using KLM predictions just what sort of user you are

1. The form of Fitts' law used with the original KLM is $K\log_2 (D/S + 0.5)$, where D is the distance to the target and S is the target size. We will use this form for calculations in this subsection, but revert to the form $a + b\log_2(D/S + 1)$ in the next subsection when we consider Buxton's three-state model as this form was used for these experiments.

thinking about. You cannot even work out the operators and then fill in the times later, as different users may choose different methods and have different placings of **M** operators due to chunking. This sounds rather onerous, but the predictions made by KLM are only meant to be an approximation, and thus reasonable guesses about levels of expertise are enough.

Individual predictions may be interesting, but the power of KLM lies in comparison. Given several systems, we can work out the methods to perform key tasks, and then use KLM to tell us which system is fastest. This is considerably cheaper than conducting lengthy experiments (levels of individual variation would demand vast numbers of trials – see Chapter 11). Furthermore, the systems need not even exist. From a description of a proposed system, we can predict the times taken for tasks. As well as comparing systems, we can compare methods within a system. This can be useful in preparing training materials, as we can choose to teach the faster methods.

Example: Using the keystroke-level model

As an example, we compare the two methods for iconizing a window given in Section 6.7.1. One used the 'L7' function key, and the other the 'CLOSE' option from the window's pop-up menu. The latter is obtained by moving to the window's title bar, depressing the left mouse button, dragging the mouse down the pop-up menu to the 'CLOSE' option, and then releasing the mouse button. We assume that the user's hand is on the mouse to begin with, and hence only the `L7-METHOD` will require a homing operator. The operators for the two methods are as follows:

`L7-METHOD`	**H**[to keyboard] **MK**[L7 function key]
`CLOSE-METHOD`	**P**[to menu bar] **B**[LEFT down] **MP**[to option] **B**[LEFT up]

The total times are thus

```
L7-METHOD     =  0.4 + 1.35 + 0.28
              =  2.03 seconds
CLOSE-METHOD  =  1.1 + 0.1 + 1.35 + 1.1 + 0.1
              =  3.75 seconds
```

The first calculation is quite straightforward, but the second needs a little unpacking. The button presses are separate down and then up actions and thus each is only timed at 0.1 of a second, rather than 0.2 for a click, or 0.28 for typing. We have also used the simplified average 1.1 second time for the pointing task. From these predictions, we can see that the `L7-METHOD` is far faster. In Section 6.7.1, Sam's selection rule was to use the `L7-METHOD` when playing blocks. To do so, he can go on playing the game using the mouse in his right hand whilst moving his left hand over the key. Thus the real time for Sam, from when he takes his attention from the game to when the command is given, is less, 2.03 seconds minus the homing time, that is 1.63 seconds. Given the method is so fast, why does Sam not use it all the time?

Perhaps the average estimates for pointing times have biased our estimate. We can be a little more precise about the `CLOSE-METHOD` timing if we use Fitts' law instead of the average 1.1 seconds. The mouse will typically be in the middle of a 25 line high window. The title bar is 1.25 lines high. Thus the distance to target ratio for the first pointing task is 10:1. The 'CLOSE' option is four items down on the pop-up menu; hence the ratio for the second pointing task is 4:1. Thus we can calculate the pointing times:

$$\begin{aligned} \mathbf{P}[\text{to menu bar}] &= 0.1 \log_2 (10.5) &= 0.339 \\ \mathbf{P}[\text{to option}] &= 0.1 \log_2 (4.5) &= 0.217 \end{aligned}$$

With these revised timings, KLM predicts the CLOSE-METHOD will take 2.1 seconds. So, Sam's selection rule is not quite as bad as it initially seemed!

Worked exercise

Do a keystroke level analysis for opening up an application in a visual desktop interface using a mouse as the pointing device, comparing at least two different methods for performing the task. Repeat the exercise using a trackball. Consider how the analysis would differ for various positions of the trackball relative to the keyboard and for other pointing devices.

Answer

We provide a keystroke level analysis for three different methods for launching an application on a visual desktop. These methods are analyzed for a conventional one-button mouse, a trackball mounted away from the keyboard and one mounted close to the keyboard. The main distinction between the two trackballs is that the second one does not require an explicit repositioning of the hands, that is there is no time required for homing the hands between the pointing device and the keyboard.

Method 1 Double clicking on application icon

Steps	Operator	Mouse	Trackball$_1$	Trackball$_2$
1. move hand to mouse	**H**[mouse]	0.400	0.400	0.000
2. mouse to icon	**P**[to icon]	0.664	1.113	1.113
3. double click	**2B**[click]	0.400	0.400	0.400
4. return to keyboard	**H**[kbd]	0.400	0.400	0.000
Total times		1.864	2.313	1.513

Method 2 Using an accelerator key

Steps	Operator	Mouse	Trackball$_1$	Trackball$_2$
1. move hand to mouse	**H**[mouse]	0.400	0.400	0.000
2. mouse to icon	**P**[to icon]	0.664	1.113	1.113
3. click to select	**B**[click]	0.200	0.200	0.200
4. pause	**M**	1.350	1.350	1.350
5. return to keyboard	**H**[kbd]	0.400	0.400	0.000
6. press accelerator	**K**	0.200	0.200	0.200
Total times		3.214	3.663	2.763

Method 3 Using a menu

Steps	Operator	Mouse	Trackball_1	Trackball_2
1. move hand to mouse	**H**[mouse]	0.400	0.400	0.000
2. mouse to icon	**P**[to icon]	0.664	1.113	1.113
3. click to select	**B**[click]	0.200	0.200	0.200
4. pause	**M**	1.350	1.350	1.350
5. mouse to file menu	**P**	0.664	1.113	1.113
6. pop-up menu	**B**[down]	0.100	0.100	0.100
7. drag to open	$\mathbf{P}_{\text{drag}}$	0.713	1.248	1.248
8. release mouse	**B**[up]	0.100	0.100	0.100
9. return to keyboard	**H**[kbd]	0.400	0.400	0.000
Total times		4.591	6.024	5.224

Card, Moran and Newell empirically validated KLM against a range of systems, both keyboard and mouse based, and a wide selection of tasks. The predictions were found to be remarkably accurate (an error of about 20%). KLM is thus one of the few models capable of giving accurate quantitative predictions about performance. However, the range of applications is correspondingly small. It tells us a lot about the microinteraction, but not about the larger-scale dialog.

However, we have seen that one has to be quite careful, as the approximations one makes can radically change the results – KLM is a guide, not an oracle. One should also add a word of caution about the assumption that fastest is best. There are certainly situations where this is so, for example highly repetitive tasks such as telephony or data entry. However, even expert users will often not use the fastest method. For example, the expert may have a set of general-purpose, non-optimal methods, rather than a few task-specific methods.

6.10.2 Three-state model

In Chapter 2, we saw that a range of pointing devices exists in addition to the mouse. Often these devices are considered logically equivalent, if the same inputs are available *to the application*. That is, so long as you can select a point on the screen, they are all the same. However, these different devices – mouse, trackball, light pen – feel very different. Although the devices are similar from the application's viewpoint, they have very different sensory–motor characteristics.

Buxton has developed a simple model of input devices [33], the *three-state model*, which captures some of these crucial distinctions. He begins by looking at a mouse. If you move it with no buttons pushed, it normally moves the mouse cursor about. This tracking behaviour is termed state 1. Depressing a button over an icon and then moving the mouse will often result in an object being dragged about. This he calls state 2 (see Figure 6.1).

If instead we consider a light pen with a button, it behaves just like a mouse when it is touching the screen. When its button is not depressed, it is in state 1, and when its button is down, state 2. However, the light pen has a third state, when the light pen is not touching the screen. In this state the system cannot track the light pen's position. This state is called state 0 (see Figure 6.2).

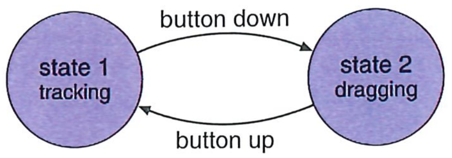

Figure 6.1 Mouse transitions: states 1 and 2

A touchscreen is like the light pen with no button. While the user is not touching the screen, the system cannot track the finger – that is, state 0 again. When the user touches the screen, the system can begin to track – state 1. So a touchscreen is a state 0–1 device whereas a mouse is a state 1–2 device. As there is no difference between a state 0–2 and a state 0–1 device, there are only the three possibilities we have seen. The only additional complexity is if the device has several buttons, in which case we would have one state for each button: 2_{left}, 2_{middle}, 2_{right}.

One use of this classification is to look at different pointing tasks, such as icon selection or line drawing, and see what state 0–1–2 behaviour they require. We can then see whether a particular device can support the required task. If we have to use an inadequate device, it is possible to use keyboard keys to add device states. For example, with a touchscreen, we may nominate the escape key to be the 'virtual' mouse button whilst the user's finger is on the screen. Although the mixing of keyboard and mouse keys is normally a bad habit, it is obviously necessary on occasions.

At first, the model appears to characterize the states of the device by the inputs available to the system. So, from this perspective, state 0 is clearly different from states 1 and 2. However, if we look at the state 1–2 transaction, we see that it is symmetric with respect to the two states. In principle there is no reason why a program should not decide to do simple mouse tracking whilst in state 2 and drag things about in state 1. That is, there is no reason until you want to type something! The way we can tell state 1 from state 2 is by the activity of the *user*. State 2 requires a button to be pressed, whereas state 1 is one of relative relaxation (whilst still requiring hand–eye coordination for mouse movement). There is a similar difference in tension between state 0 and state 1.

It is well known that Fitts' law has different timing constants for different devices. Recall that Fitts' law says that the time taken to move to a target of size S at a distance D is:

$$a + b\log_2(D/S + 1)$$

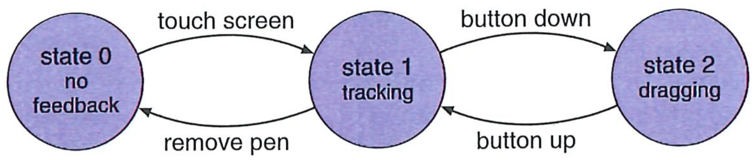

Figure 6.2 Light pen transitions: three states

The constants a and b depend on the particular pointing device used and the skill of the user with that device. However, the insight given by the three-state model is that these constants also depend on the device state. In addition to the timing, the final accuracy may be affected.

These observations are fairly obvious for state 0–1 devices. With a touchscreen, or light pen, a cursor will often appear under the finger or pen when it comes in contact with the screen. The accuracy with which you can move the cursor around will be far greater than the accuracy with which you can point in the first place. Also it is reasonable to expect that the Fitts' law constant will be different, although not so obvious which will be faster.

There is a similar difference between states 1 and 2. Because the user is holding a button down, the hand is in a state of tension and thus pointing accuracy and speed may be different. Experiments to calculate Fitts' law constants in states 1 and 2 have shown that these differences do exist [146]. Table 6.2 shows the results obtained for a mouse and trackball.

We can recalculate the KLM prediction for the `CLOSE-METHOD` using these data. Recall that the method had two pointing operators, one to point to the window's title bar (with a distance to target size ratio of 10:1), the second to drag the selection down to 'CLOSE' on the pop-up menu (4:1). Thus the first pointing operator is state 1 and the second is state 2. The times are thus

Mouse

$$\begin{aligned} \mathbf{P}[\text{to menu bar}] &= -107 + 223 \log_2(11) &&= 664 \text{ ms} \\ \mathbf{P}[\text{to option}] &= 135 + 249 \log_2(5) &&= 713 \text{ ms} \end{aligned}$$

Trackball

$$\begin{aligned} \mathbf{P}[\text{to menu bar}] &= 75 + 300 \log_2(11) &&= 1113 \text{ ms} \\ \mathbf{P}[\text{to option}] &= -349 + 688 \log_2(5) &&= 1248 \text{ ms} \end{aligned}$$

giving a further revised time for the `CLOSE-METHOD` of 2.93 seconds using a mouse and 3.91 seconds using a trackball.

Table 6.2 Fitts' law coefficients (after MacKenzie, Sellen and Buxton [146])

	Device	a (ms)	b (ms/bit)
Pointing (state 1)			
	Mouse	−107	223
	Trackball	75	300
Dragging (state 2)			
	Mouse	135	249
	Trackball	−349	688

6.11 Cognitive architectures

The formalisms we have seen so far have some implicit or explicit model of how the user performs the cognitive processing involved in carrying out a task. For instance, the concept of taking a problem and solving it by divide and conquer using subgoals is central to GOMS. CCT assumes the distinction between long- and short-term memory, with production rules being stored in long-term memory and 'matched' against the contents of short-term (or working) memory to determine which 'fire'. The values for various motor and mental operators in KLM were based on the Model Human Processor (*MHP*) architecture of Card, Moran and Newell [37]. Another common assumption which we have not discussed in this chapter is the distinction between linguistic levels – semantic, syntactic and lexical – as an architectural model of the user's understanding.

In Chapter 1, we discussed some of these architectural descriptions of the user as an information-processing machine. Our emphasis in this section will be to describe a couple more architectural models that are quite distinct from those described in Chapter 1 and assumed in the earlier parts of this chapter. Here we will see that the architectural assumptions are central to the description of the cognitive modelling that these approaches offer.

There are interesting differences of emphasis between these architectural models and the previous models. The hierarchical and linguistic notations tend to assume perfect dialog on the user's part. They may measure the complexity of that perfect dialog, but tend not to consider diversions from the optimal command sequences. However, for the architectural models in this section the prediction and understanding of error is central to their analyses.

6.11.1 The problem space model

Rational behaviour is characterized as behaviour which is intended to achieve a specific goal. This element of rationality is often used to distinguish between intelligent and machine-like behaviour. In the field of artificial intelligence (AI), a system exhibiting rational behaviour is referred to as a *knowledge-level* system. A knowledge-level system contains an *agent* behaving in an environment. The agent has knowledge about itself and its environment, includings its own goals. It can perform certain actions and sense information about its changing environment. As the agent behaves in its environment, it changes the environment and its own knowledge. We can view the overall behaviour of the knowledge-level system as a sequence of environment and agent states as they progress in time. The goal of the agent is characterized as a preference over all possible sequences of agent/environment states.

Contrast this rational behaviour with another general computational model for a machine, which is not rational. In computer science it is common to describe a problem as the search through a set of possible states from some initial state to a desired state. The search proceeds by moving from one state to another possible state by means of operations or actions, the ultimate goal of which is to arrive at one of the desired states. This very general model of computation is used in the ordinary task of the programmer. Once she has identified a problem and a means of arriving at the solution to the problem (the algorithm), the programmer then represents the problem and algorithm in a programming language which can be executed on a machine to reach the desired state. The architecture of the machine only allows the definition of the search or *problem space* and the actions which can occur to traverse that space. Termination is also assumed to happen once the desired state is reached. Notice that the machine does not have the ability to formulate the problem space and its solution, mainly because it has no idea of the goal. It is the job of the programmer to understand the goal and so define the machine to achieve it.

We can adapt the state-based computational model of a machine in order to realize the architecture of a knowledge-level system. The new computational model is the *problem space* model, based on the problem-solving work of Newell and Simon at Carnegie–Mellon University (see Chapter 1). A problem space consists of a set of states and a set of operations that can be performed on the states. Behaviour in a problem space is a two-step process. First, the current operator is chosen based on the current state and then it is applied to the current state to achieve the new state. The problem space must represent rational behaviour, and so it must characterize the goal of the agent. A problem space represents a goal by defining the desired states as a subset of all possible states. Once the initial state is set, the task within the problem space is to find a sequence of operations that form a path within the state space from the initial state to one of the desired states, whereupon successful termination occurs.

From the above description, we can highlight four different activities that occur within a problem space: goal formulation, operation selection, operation application and goal completion. The relationship between these problem space processes and knowledge-level activity is key. Perception which occurs at the knowledge level is performed by the goal formulation process, which creates the initial state based on observations of the external environment. Actions at the knowledge level are operations in the problem space which are selected and applied. The real knowledge about the agent and its environment and goals is derived from the state/operator information in the problem space. Because of the goal formulation process, the set of desired states indicates the knowledge-level goal within the problem space. The operation selection process selects the appropriate operation at a given point in time because it is deemed the most likely to transform the state in the problem space to one of the desired states; hence rational behaviour is implied.

The cycle of activity within the problem space is as follows. Some change in the external environment which is relevant to the goal of the agent is sensed by the goal formulation process, which in turn defines the set of desired states for the agent and its initial state for the following task. The operation selection process suggests an operation which can act on that state and transform it 'closer' to a desired state. The operation application process executes the operation, changing the current state and surrounding environment. If the new state is a desired state, then the goal has been achieved and the goal completion process reverts the agent to inactive.

The real power of the problem space architecture is in recursion. The activity of any of the processes occurs only when the knowledge it needs to complete its chore is immediately available. For example, to decide which operation is most likely to lead to a desired state, the problem space will need to know things about its current state and that of the environment. If that information is not immediately available, then activity cannot continue. In that case, another problem space is invoked with the goal of finding out the information that was needed by the parent problem space. In this way, we can see the evolution of problems spaces as a stack-like structure, new spaces being invoked and placed on the problem space stack only to be popped off the stack once they achieve their goal.

Though the problem space model described briefly above is not directly implementable, it is the basis for at least one executable cognitive architecture, called Soar. We do not discuss the details of Soar's implementation; the interested reader is referred to Laird, Newell and Rosenbloom [133]. An interesting application of the Soar implementation of problem spaces has been done by Young and colleagues on *programmable user models* (or *PUMs*) [266]. Given a designer's description of an intended procedure or task that is to be carried out with an interactive system, an analysis of that procedure produces the knowledge that would be necessary and available for any user attempting the task. That knowledge is encoded in the problem space architecture of Soar, producing a 'programmed' user model (the PUM) to accomplish the goal of performing the task. By executing the PUM, the stacking and unstacking of problem spaces needed to accomplish the goal can be analyzed to measure the cognitive load of the intended procedure. More importantly, if the PUM cannot achieve the goal because it cannot find some knowledge necessary to complete the task, this indicates to the designer that there was a problem with the intended design. In this way, erroneous behaviour can be predicted before implementation.

6.11.2 Interacting cognitive subsystems

Barnard has proposed a very different cognitive architecture, called *interacting cognitive subsystems* (ICS) [17, 18, 19]. ICS provides a model of perception, cognition and action, but unlike other cognitive architectures, it is not intended to produce a description of the user in terms of sequences of actions that he performs. ICS provides a more holistic view of the user as an information-processing machine. The emphasis is on determining how easy particular procedures of action sequences become as they are made more automatic within the user.

ICS attempts to incorporate two separate psychological traditions within one cognitive architecture. On the one hand is the architectural and general-purpose information-processing approach of short-term memory research. On the other hand is the computational and representational approach characteristic of psycholinguistic research and AI problem-solving literature.

The architecture of ICS is built up by the coordinated activity of nine smaller subsystems: five peripheral subsystems are in contact with the physical world and four are central, dealing with mental processes. Each subsystem has the same generic structure. A subsystem is described in terms of its typed inputs and outputs along with a memory store for holding typed information. It has transformation functions for processing the input and producing the output and permanently stored information. Each of the nine subsystems is specialized for handling some aspect of external or internal processing. For example, one peripheral subsystem is the visual system for describing what is seen in the world. An example of a central subsystem is one for the processing of propositional information, capturing the attributes and identities of entities and their relationships with each other (a particular example is that propositional information represents '"knowing" that a particular word has four syllables, begins with "P" and refers to an area in central London' [18]).

ICS is another example of a general cognitive architecture which can be applied to interactive design. One of the features of ICS is its ability to explain how a user proceduralizes action. Remember in the discussion of CCT we distinguished between novice and expert use of an interactive system. Experts can perform complicated sequences of actions as if without a thought, whereas a novice user must contemplate each and every move (if you do not believe this distinction is accurate, observe users at an automatic teller machine and see if you can tell the expert from the novice). The expert recognizes the task situation and recalls a 'canned' procedure of actions which, from experience, results in the desired goal being achieved. They do not have to think beyond the recognition of the task and consequent invocation of the correct procedure. Such proceduralized behaviour is much less prone to error. A good designer will aid the user in proceduralizing his interaction with the system and will attempt to design an interface which suggests to the user a task for which he already has a proceduralized response. It is for this reason that ICS has been suggested as a design tool which can act as an expert system to advise a designer in developing an interface.

6.12 Summary

In this chapter, we have discussed a wide selection of models of the users of interactive systems, including socio-technical and systems models and cognitive models. Socio-technical models focus on representing both the human and technical sides of the system in parallel to reach a solution which is compatible with each. SSM models the organization, of which the user is part, as a system. Participatory design sees the user as active not only in using the technology but in designing it. Cognitive models attempt to represent the users as they interact with a system, modelling aspects of their understanding, knowledge, intentions or processing. We divided cognitive models into three categories. The first described the hierarchical structuring of the user's task and goal structures. The GOMS model and CCT were examples of cognitive models in this category. The second category was concerned with linguistic and grammatical models which emphasized the user's understanding of the user–system dialog. BNF and TAG were described as examples in this category. Most of these cognitive models have focused on the execution activity of the user, neglecting their perceptive ability and how that might affect less planned and natural interaction strategies. The third category of cognitive models was based on the more solid understanding of the human motor system, applicable in situations where the user does no planning of behaviour and executes actions automatically. KLM was used to provide rough measures of user performance in terms of execution times for basic sequences of actions. Buxton's three-state model for pointing devices allowed for a finer distinction between execution times than with KLM. We concluded this chapter with a discussion of cognitive architectures, the assumptions of which form the foundation for any cognitive models. In addition to the basic architectural distinction between long- and short-term memory, we discussed two other cognitive architectures – the problem space model and ICS – which apply different assumptions to the analysis of interactive system.

Exercises

6.1 A group of universities has decided to collaborate to produce an information system to help potential students find appropriate courses. The system will be distributed free to schools and careers offices on CD-ROM and will provide information about course contents and requirements, university and local facilities, fees and admissions procedures. Identify the main stakeholders for this system, categorize them and describe them and their activities, currently and with regard to the proposed system.

6.2 For the scenario proposed above:

- ❑ Produce a rich picture showing the problem situation (you can use any format that you find helpful).
- ❑ Produce a root definition, using CATWOE, of the system from the viewpoint of the university.
- ❑ What transformations or activities are required to make sure that the root definition is supported?

6.3 Recall the CCT description of the rule INSERT-SPACE-2 discussed in Section 6.7.2:

```
(INSERT-SPACE-2
IF (AND (TEST-GOAL insert space)
        (TEST-CURSOR %LINE %COL) )
THEN (  (DO-KEYSTROKE 'I')
        (DO-KEYSTROKE SPACE)
        (DO-KEYSTROKE ESC)
        (DELETE-GOAL insert space) ))
```

As we discussed, this is already proceduralized, that is the rule is an expert rule. Write new 'novice' rules where the three keystrokes are not proceduralized. That is, you should have separate rules for each keystroke and suitable goals (such as GET-INTO-INSERT-MODE) to fire them.

6.4 One of the assumptions underlying the programmable user model approach is that it is possible to provide an algorithm to describe the user's behaviour in interacting with a system. Taking this position to the extreme, choose some common task with a familiar interactive system (for example, creating a column of numbers in a spreadsheet and calculating their sum, or any other task you can think of) and describe the algorithm needed by the user to accomplish this task. Write the description in pseudocode. Does this exercise suggest any improvements in the system?

Recommended reading

- ❑ K. D. Eason, *Information Technology and Organizational Change*, Taylor and Francis, 1988.
  Clear coverage of socio-technical and organizational issues in design.
- ❑ P. B. Checkland, *Systems Thinking, Systems Practice*, John Wiley, 1981.
  The standard text which covers soft systems methodology in detail.
- ❑ S. K. Card, T. P. Moran and A. Newell, *The Psychology of Human Computer Interaction*, Lawrence Erlbaum, 1983.
  A classic text in this field of cognitive models, in which the basic architectural assumptions of the Model Human Processor architecture are explained as well as the GOMS model and KLM.
- ❑ S. Bovair, D. E. Kieras and P. G. Polson, The acquisition and performance of text-editing skill: a cognitive complexity analysis. *Human–Computer Interaction*, Vol. 5, No. 1, pp. 1–48, 1990.
  This article provides a definitive description of CCT by means of an extended example. The authors also provide the definition of various style rules for writing CCT descriptions to distinguish, for example, between novice and expert users.
- ❑ F. Schiele and T. Green, HCI formalisms and cognitive psychology: the case of task–action grammars. In M. D. Harrison and H. W. Thimbleby, editors, *Formal methods in Human–Computer Interaction*, chapter 2, Cambridge University Press, 1990.
  A good description of TAG with several extended examples based on the Macintosh interface. The authors provide a good comparative analysis of TAG versus other cognitive modelling techniques.
- ❑ A. Newell, G. Yost, J. E. Laird, P. S. Rosenbloom and E. Altmann, Formulating the problem-space computational model. In R. F. Rashid, editor, *CMU Computer Science: a 25th Anniversary Commemorative*, chapter 11, ACM Press, 1991.
  The description of the problem space cognitive architecture was informed by this article, which also contains references to essential work on the Soar platform.