# A Guide to GOMS Model Usability Evaluation using NGOMSL

**David Kieras**

**University of Michigan**

David Kieras, Artificial Intelligence Laboratory
Electrical Engineering and Computer Science Department
University of Michigan
Advanced Technology Laboratory Building
1101 Beal Avenue, Ann Arbor, MI 48109-2110
Phone: (313) 763-6739; Fax: (313) 763-1260
Email: kieras@eecs.umich.edu
Anonymous ftp for documents: ftp.eecs.umich.edu people/kieras

**Spring, 1996**

# 1. INTRODUCTION

## 1.1 Overview

### Engineering Models for Usable Interface Design

The standard accepted technique for developing a usable system, empirical user testing, is based on iterative testing and design revision using actual users to test the system and help identify usability problems. It is widely agreed that this approach, inherited from Human Factors, does indeed work when carefully applied (Landauer, 1995). However, Card, Moran, & Newell (1983) have argued, and many HCI researchers have agreed (e.g., see Butler, Bennett, Polson, & Karat, 1989), that empirical user testing is too slow and expensive for modern software development practice, especially when difficult-to-get domain experts are the target user group. One response has been the development of "discount" or "inspection" methods for assessing the usability of an interface design quickly and at low cost (Nielsen & Mack, 1994). However, another response, which has been evolving since the seminal Card, Moran, and Newell work, is the concept of *engineering models* for usability. Analogously to the models used in other engineering disciplines, engineering models for usability produce quantitative predictions of how well humans will be able to perform tasks with a proposed design. Such predictions can be used as a surrogate for actual empirical user data, making it possible to iterate through design revisions and evaluations much more rapidly. Furthermore, unlike purely empirical assessments, an engineering model for an interface design can capture the essence of the design in an inspectable representation, making it easier to reuse successful design insights in the future.

The overall scheme for using engineering models in the user interface design process is as follows: Following an initial task analysis and proposed first interface design, the interface designer would then use an engineering model as applicable to find the usability problems in the interface. However, because there are other aspects of usability that are poorly understood, some form of user testing is still required to ensure a quality result. Only after dealing with design problems revealed by the engineering model would the designer then go on to user testing. If the user testing reveals a serious problem, the design might have to be fundamentally revised, but again the engineering models will help refine the redesign quickly. Thus the slow and expensive process of user testing is reserved for those aspects of usability that can only be addressed at this time by empirical trials. If engineering models can be fully developed and put into use, then the designer's creativity and development resources can be more fully devoted to more challenging design problems, such as devising entirely new interface concepts or approaches to the design problem at hand.

### The GOMS Model

The major extant form of engineering model for interface design is the GOMS model, first proposed by Card, Moran, and Newell (1983). A GOMS model is a description of the knowledge that a user must have in order to carry out tasks on a device or system; it is a representation of the "how to do it" knowledge that is required by a system in order to get the intended tasks accomplished. The acronym GOMS stands for Goals, Operators, Methods, and Selection Rules. Briefly, a GOMS model consists of descriptions of the Methods needed to accomplish specified Goals. The Methods are a series of steps consisting of Operators that the user performs. A Method may call for sub-Goals to be accomplished, so the Methods have a hierarchical structure. If there is more than one Method to accomplish a Goal, then Selection Rules choose the appropriate Method depending on the context. Describing the Goals, Operators, Methods, and Selection Rules for a set of tasks in a formal way constitutes doing a GOMS analysis, or constructing a GOMS model.

John & Kieras (1994) describe the current family of GOMS models and the associated techniques for predicting usability, and list many successful applications of GOMS to practical design problems. The simplest form of GOMS model is the Keystroke-Level Model, first described by Card, Moran, and Newell (1980), in which

task execution time is predicted by the total of the times for the elementary keystroke-level actions required to perform the task. The most complex is CPM-GOMS, developed by Gray, John, and Atwood (1993), in which the sequential dependencies between the user's perceptual, cognitive, and motor processes are mapped out in a schedule chart, whose critical path predicts the execution time.

In between these two methods is the method presented in this article, NGOMSL, in which learning time and execution time are predicted based on a program-like representation of the procedures that the user must learn and execute to perform tasks with the system. NGOMSL is an acronym for **N**atural **GOMS L**anguage, which is a structured natural language used to represent the user's methods and selection rules. NGOMSL models thus have an explicit representation of the user's methods, which are assumed to be strictly sequential and hierarchical in form. The execution time for a task is predicted by simulating the execution of the methods required to perform the task. Each NGOMSL statement is assumed to require a small fixed time to execute, and any operators in the statement, such as a keystroke, will then take additional time depending on the operator. The time to learn how to operate the interface can be predicted from the length of the methods, and the amount of transfer of training from the number of methods or method steps previously learned. Thus estimating times for learning and execution both require counting the number of NGOMSL statements involved; details on this process will be provided in this article.

One important feature of NGOMSL models is that the "how to do it" knowledge is described in a form that can actually be executed – the analyst, or an appropriately programmed computer, can go through the GOMS methods, executing the described actions, and actually carry out the task. A GOMS model is also a way to characterize a set of design decisions from the point of view of the user, which can make it useful during, as well as after, design. It is also a description of what the user must learn, and so can act as a basis for training and reference documentation.

NGOMSL is based on the cognitive modeling of human-computer interaction by Kieras and Polson (Kieras & Polson, 1985; Bovair, Kieras, & Polson, 1990). As summarized by John and Kieras (1994), NGOMSL is useful for many desktop computing situations in which the user's procedures are usefully approximated as being hierarchical and sequential.

**Strengths and Limitations of GOMS Models**

It is important to be clear on what GOMS models can and cannot do; see John and Kieras (1994) for more discussion.

***GOMS starts* after *a task analysis.*** In order to apply the GOMS technique, the designer (or interface analyst, hereafter just referred to as the designer) must conduct a task analysis to identify what goals the user will be trying to accomplish. The designer can then express in a GOMS model how the user can accomplish these goals with the system being designed. Thus, GOMS modeling does not replace the most critical process in designing a usable system, that of understanding the user's situation, working context, and goals. Approaches to this stage of interface design have been presented in sources such Gould (1988), Diaper (1989), Kirwan and Ainsworth (1992), and Kieras (in press).

***GOMS represents only the procedural aspects of usability.*** GOMS models can predict the *procedural* aspects of usability; these concern the amount, consistency, and efficiency of the procedures that users must follow. Since the usability of many systems depends heavily on the simplicity and efficiency of the procedures, the narrowly focused GOMS model has considerable value in guiding interface design. The reason why GOMS models can predict these aspects of usability is that the methods for accomplishing user goals tend to be tightly constrained by the design of the interface, making it possible to construct a GOMS model given just the interface design, prior to any prototyping or user testing.

Clearly, there are other important aspects of usability that are not related to the procedures entailed by the interface design. These concern both lowest-level perceptual issues like the legibility of typefaces on CRTs, and

also very high-level issues such as the user's conceptual knowledge of the system, e.g., whether the user has an appropriate "mental model" (e.g. Kieras & Bovair, 1984), or the extent to which the system fits appropriately into an organization (see John & Kieras, 1994). The lowest-level issues are dealt with well by standard human factors methodology, while understanding the higher-level concerns is currently a matter of practitioner wisdom and the higher-level task analysis techniques. Considerably more research is needed on the higher-level aspects of usability, and tools for dealing with the corresponding design issues are far off. For these reasons, great attention must still be given to the task analysis, and some user testing will still be required to ensure a high-quality user interface.

***GOMS models are practical and effective.*** There has been a widespread belief that constructing and using GOMS models is too time-consuming to be practical (e.g., Lewis & Rieman, 1994). However, the many cases surveyed by John & Kieras (1994) make clear that members of the GOMS family have been applied in many practical situations and were often very time- and cost-effective. A possible source of confusion is that the development of the GOMS modeling techniques has involved validating the analysis against empirical data. However, once the technique has been validated and the relevant parameters estimated, no empirical data collection or validation should be needed to apply a GOMS analysis during practical system design, enabling usability evaluations to be obtained much faster than user testing techniques. However, the calculations required to derive the predictions are tedious and mechanical; at the time of this writing, computer-based tools for developing and using GOMS models are under development (e.g, Byrne, Wood, Sukaviriya, Foley, & Kieras, 1994; Kieras, Wood, Abotel, & Hornof, 1995).

**What is a GOMS Task Analysis?**

Describing the Goals, Operators, Methods, and Selection Rules for a set of tasks in a relatively formal way constitutes doing a GOMS analysis. The person who is performing such an analysis is referred to as "the analyst" in this Guide.

Carrying out a GOMS analysis involves defining and then describing in a formal notation the user's Goals, Operators, Methods, and Selection Rules. Most of the work seems to be in defining the Goals and Methods. That is, the Operators are mostly determined by the hardware and lowest-level software of the system, such as whether it has a mouse, for example. Thus the Operators are fairly easy to define. The Selection Rules can be subtle, but usually they are involved only when there are clear multiple methods for the same goal. In a good design, it is clear when each method should be used, so defining the Selection Rules is (or should be) relatively easy as well.

Identifying and defining the user's goals is often difficult, because you must examine the task that the user is trying to accomplish in some detail, often going beyond just the specific system to the context in which the system is being used. This is especially important in designing a new system, because a good design is one that fits not just the task considered in isolation, but also how the system will be used in the user's job context. As mentioned above, GOMS modeling starts with the results of a task analysis that identifies the user's goals. For brevity, task analysis per se will not be discussed further here; excellent sources are Gould (1988), .Diaper (1989), and Kirwan and Ainsworth (1992); see Kieras (in press) for an overview.

Once a Goal is defined, the corresponding method can be simple to define because it is simply the answer to the question "how do you do it on this system?" The system design itself largely determines what the methods are.

One critical process involved in doing a GOMS analysis is deciding what and what *not* to describe. The mental processes of the user can be of incredible complexity; trying to describe all of them would be hopeless. However, many of these complex processes have nothing to do with the design of the interface, and so do not need to be analyzed. For example, the process of reading is extraordinarily complex; but usually, design choices for a user interface can be made without any detailed consideration of how the reading process works. We can treat the user's reading mechanisms as a "black box" during the interface design. We may want to know *how much* reading has to

be done, but rarely do we need to know *how* it is done. So, we will need to describe when something is read, and why it is read, but we will not need to describe the actual processes involved. A way to handle this in a GOMS analysis is to "bypass" the reading process by representing it with a "dummy" or "place holder" operator. This is discussed more below. But making the choices of what to bypass is an important, and sometimes difficult, part of the analysis.

## 1.2 Example of GOMS Analysis Results

Before presenting the details of the GOMS modeling methodology, it is useful to examine a sample GOMS analysis and see how it can capture an important aspect of interface "consistency." The tasks and systems are some file manipulation tasks in PC-DOS and Macintosh Finder. The set of user goals considered are:

- delete a file
- move a file
- delete a directory
- move a directory

The example consists of a list of methods for each system, expressed in the NGOMSL notation introduced in detail later in this Guide. One of the virtues of this notation is that it is pretty comprehensible without having studied its formal definition first. This example is intended just to give the overall flavor of what a GOMS model looks like, so no detail or explanation of the notation will be given at this time.

**GOMS Model for Macintosh Finder**

There is a specific method for accomplishing each one of the user goals under consideration. Notice how each method is simply an explicit step-by-step description of what the user has to do in order to accomplish the goal.

```
Method for goal: delete a file.
    Step 1. Accomplish goal: drag file to trash.
    Step 2. Return with goal accomplished.

Method for goal: move a file.
    Step 1. Accomplish goal: drag file to destination.
    Step 2. Return with goal accomplished.

Method for goal: delete a directory.
    Step 1. Accomplish goal: drag directory to trash.
    Step 2. Return with goal accomplished.

Method for goal: move a directory.
    Step 1. Accomplish goal: drag directory to destination.
    Step 2. Return with goal accomplished.
```

We can see from the above methods that they all have a simple pattern; it doesn't matter whether a directory or a file is being manipulated. So we can replace the above four methods with only two generalized methods, one for deleting and one for moving:

```
Method for goal: delete an object.
    Step 1. Accomplish goal: drag object to trash.
    Step 2. Return with goal accomplished.
```

```
Method for goal: move an object.
    Step 1. Accomplish goal: drag object to destination.
    Step 2. Return with goal accomplished.
```

In addition to the specific moving and deleting methods, there is a general submethod corresponding to the drag operation; this is the basic method used in most of the Macintosh Finder file manipulations. It is called like a subroutine by the above methods.

```
Method for goal: drag item to destination.
    Step 1. Locate icon for item on screen.
    Step 2. Move cursor to item icon location.
    Step 3. Hold mouse button down.
    Step 4. Locate destination icon on screen.
    Step 5. Move cursor to destination icon.
    Step 6. Verify that destination icon is reverse-video.
    Step 7. Release mouse button.
    Step 8. Return with goal accomplished.
```

**GOMS Model for PC-DOS**

There are a large number of specific methods, and some of the user goals, such as moving a file, are accomplished by calling other methods. Notice also that there is no generalization over files and directories, because the PC-DOS command set forces us to use very different methods for these two types of objects. Each of these methods calls a general submethod for entering and executing a specified command.

```
Method for goal: delete a file.
    Step 1. Recall that command verb is "ERASE".
    Step 2. Think of directory name and file name and retain as first filespec.
    Step 4. Accomplish goal: enter and execute a command.
    Step 6. Return with goal accomplished.

Method for goal: move a file.
    Step 1. Accomplish goal: copy a file.
    Step 2. Accomplish goal: delete a file.
    Step 3. Return with goal accomplished.

Method for goal: copy a file.
    Step 1. Recall that command verb is "COPY".
    Step 2. Think of source directory name and file name and retain as first
            filespec.
    Step 3. Think of destination directory name and file name and retain as
            second filespec.
    Step 4. Accomplish goal: enter and execute a command.
    Step 5. Return with goal accomplished.

Method for goal: delete a directory.
    Step 1. Accomplish goal: delete all files in the directory.
    Step 2. Accomplish goal: remove a directory.
    Step 3. Return with goal accomplished.
```

```
Method for goal: delete all files in a directory.
    Step 1. Recall that command verb is "ERASE".
    Step 2. Think of directory name.
    Step 3. Retain directory name and "*.*" as first filespec.
    Step 4. Accomplish goal: enter and execute a command.
    Step 5. Return with goal accomplished.

Method for goal: remove a directory
    Step 1. Recall that command verb is "RMDIR".
    Step 2. Think of directory name and retain as first filespec.
    Step 3. Accomplish goal: enter and execute a command.
    Step 4. Return with goal accomplished.

Method for goal: move a directory.
    Step 1. Accomplish goal: copy a directory.
    Step 2. Accomplish goal: delete a directory.
    Step 3. Return with goal accomplished.

Method for goal: copy a directory.
    Step 1. Accomplish goal: create a directory.
    Step 2. Accomplish goal: copy all the files in a directory.
    Step 3. Return with goal accomplished.

Method for goal: create a directory.
    Step 1. Recall that command verb is "MKDIR".
    Step 2. Think of directory name and retain as first filespec.
    Step 3. Accomplish goal: enter and execute a command.
    Step 4. Return with goal accomplished.

Method for goal: copy all files in a directory.
    Step 1. Recall that command verb is "COPY".
    Step 2. Think of directory name.
    Step 3. Retain directory name and "*.*" as first filespec.
    Step 4. Think of destination directory name.
    Step 5. Retain destination directory name and "*.*" as second filespec.
    Step 6. Accomplish goal: enter and execute a command.
    Step 7. Return with goal accomplished.
```

The following general submethods are called by the above methods. They reflect the basic consistency of the command structure, in which each command consists of a verb followed by one or two file specifications.

```
Method for goal: enter and execute a command.
Entered with strings for a command verb and one or two filespecs.
    Step 1. Type command verb.
    Step 2. Accomplish goal: enter first filespec.
    Step 3. Decide: If no second filespec, goto 5.
    Step 4. Accomplish goal: enter second filespec.
    Step 5. Verify command.
    Step 6. Type "<CR>".
    Step 7. Return with goal accomplished.
```

```
Method for goal: enter a filespec.
Entered with directory name and file name strings.
    Step 1. Type space.
    Step 2. Decide: If no directory name, goto 5.
    Step 3. Type "\".
    Step 4. Type directory name.
    Step 5. Decide: If no file name, return with goal accomplished.
    Step 6. Type file name.
    Step 7. Return with goal accomplished.
```

**GOMS Comparison of Macintosh Finder and PC-DOS**

Clearly there is a substantial difference in the number and length of methods between these two systems. The Macintosh Finder, in its generalized form requires only 3 methods to accomplish these user goals, involving a total of only 18 steps. To accomplish the same goals in PC-DOS requires 12 methods with a total of 68 steps. Thus we have a clear characterization of the extreme consistency of the Macintosh Finder compared to PC-DOS; only a few methods are required to accomplish a variety of file manipulation goals. A major value of a GOMS model is its ability to characterize, and even quantify, this property of *method consistency*.

The Guide describes below how these method descriptions can be used to derive predictions of learning and execution time. The user must learn these step-by-step methods in order to learn how to perform these tasks. According to research results, the learning time is linear with the number of steps. The execution time can be predicted as in the Keystroke-Level Model, as a total of the individual operator times.

### 1.3 Organization of this Guide

This presentation of the NGOMSL methodology supersedes earlier presentations (Kieras, 1988; 1994). The remainder of this Guide is organized as follows: Section 2 defines the parts of a GOMS model in terms of the NGOMSL notation. Section 3 discusses some of the general issues that underlie the approach. Section 4 presents the procedure for constructing a GOMS model, along with an extended example. Section 5 explains how to use a GOMS model evaluation of a design for predicting human performance, and how a revised design and documentation can be based on the model.

## 2. DEFINITIONS AND A NOTATION FOR GOMS MODELS

This section defines each component of a GOMS model in more detail than Card, Moran, and Newell (1983). In addition, this section introduces a notation system, NGOMSL ("Natural GOMS Language"), which is an attempt to define a language that will allow GOMS models to be written down with a high degree of precision, but without the syntactic burden of ordinary formal languages, and that is also easy to read rather than cryptic and abbreviated. Despite the resulting verbosity and looseness, NGOMSL is close to a formal GOMS language that can be implemented as a running computer language. However, it is important to keep in mind that NGOMSL is not supposed to be an ordinary programming language for computers, but rather to have properties that are directly related to the underlying production rule models described by Kieras, Bovair, and Polson (Kieras & Polson, 1985; Polson, 1987; Kieras & Bovair, 1986; Bovair, Kieras, & Polson, 1990). So NGOMSL is supposed to represent something like "the programming language of the mind," as absurd as this sounds. The idea is that NGOMSL programs have properties that are related in straightforward ways to both data on human performance and theoretical ideas in cognitive psychology. If NGOMSL is clumsy and limited as a computer language, it is because humans

have a different architecture than computers. Thus, for example, NGOMSL does not allow complicated conditional statements, because there is good reason to believe that humans cannot process complex conditionals in a single cognitive step. If it is hard for people to do, then it should be reflected in a long and complicated NGOMSL program.

In this document, NGOMSL expressions are shown in `this typeface`. In using this notation, you may well be tempted to abbreviate it; as long as the control structure and organization is not changed, this does not present a problem in interpreting the results.

### 2.1 Goals

A goal is something that the user tries to accomplish. The analyst attempts to identify and represent the goals that typical users will have. A set of goals usually will have a hierarchical arrangement in which accomplishing a goal may require first accomplishing one or more subgoals.

A goal description is an action-object pair in the form: `<verb noun>`, such as `delete word`, or `move-by-find-function cursor`. The verb can be complicated if necessary to distinguish between methods (see below on selection rules). Any parameters or modifiers, such as where a to-be-deleted word is located, are represented in the task description (see below).

### 2.2 Operators

Operators are actions that the user executes. There is an important difference between goals and operators. Both take an action-object form, such as the goal of `revise document` and the operator of `press key`. But in a GOMS model, a goal is something to be accomplished, while an operator is just executed. This distinction is intuitively-based, and is also relative; it depends on the level of analysis.

That is, an operator is an action that we choose not to analyze into finer detail, while we normally will want to provide information on how a goal is to be accomplished. For example, we would want to describe in a GOMS model how the user is supposed to get a document revised, but we would probably take pressing a key as a primitive action that it is not necessary to further describe.

A good heuristic for distinguishing operators and goals: if you interrupt the user, and ask "what are you trying to do?" you will get in response statements of goals, not operators. Thus, you are likely to get statements like "I'm cutting and pasting this," not "I'm pressing the return key."

Typical examples:

- goals - revise document, change word, select text to be deleted
- operators - press a key, find a specific menu item on the screen

The procedure presented below for doing a GOMS analysis is based on the idea of first describing methods using very high-level operators, and then replacing these operators with methods that accomplish the corresponding goal by executing a series of lower-level operators. This process is repeated until the operators are all *primitive external operators*, chosen by the analyst, that will not be further analyzed.

**Kinds of Operators**

***External operators.*** External operators are the observable actions through which the user exchanges information with the system or other objects in the environment. These include perceptual operators, which read text from a screen, scan the screen to locate the cursor and so forth, and motor operators, such as pressing a key, or moving a mouse. External operators also include interactions with other objects in the environment, such as turning a page in a marked-up manuscript, or finding the next markup on the manuscript. The analyst usually chooses or defines the external operators depending on the system or tasks. E.g., is there a mouse on the machine? Does the user work from a marked-up paper copy of the document?

***Mental operators.*** Mental operators are the internal actions performed by the user; they are non-observed and hypothetical, inferred by the theorist or analyst. In the notation system presented here, some mental operators are "built in;" these operators correspond to the basic mechanisms of the cognitive processor, the "cognitive architecture." These are based on the production rule models described by Bovair, Kieras, and Polson 1990). These operators include actions like making a basic decision, recalling an item in Working Memory (WM), retrieving information from Long-Term Memory (LTM), or setting up a goal.

Other mental operators are defined by the analyst to represent complex mental activities (see below). Typical examples of such *analyst-defined* mental operators are determining the string to use for a find command, and determining the editing change meant by a marking on a marked-up manuscript.

***Primitive and high-level operators.*** A particular task analysis assumes a particular level of analysis which is reflected in the "grain size" of the operators. If an operator will not be decomposed into finer level, then it is a primitive operator. But if an operator will be decomposed into a sequence of lower-level, or primitive, operators, then it is a high-level operator. Specifically which operators are primitives depends on the finest grain level of analysis desired by the analyst.

Some typical primitive operators are actions like pressing a button, or moving the hand. All built-in mental operators are primitive by definition. High-level operators would be gross actions, or stand-ins for more detailed analysis, such as `LOG-INTO-SYSTEM`. The analyst recognizes that these could be decomposed, but may choose not to do so, depending on the purpose of the analysis.

**Standard Primitive External Operators**

The analyst defines the primitive motor and perceptual operators based on the elementary actions needed by the system being analyzed. These correspond directly to the physical and some of the mental operators used in the Keystroke-Level Model. Some typical examples and their Keystroke-Level Model equivalents:

```
Home hand to mouse (H)
Press <key name> (K)
Type a string of characters <string> (T)
Press or release mouse button (B)
Click mouse button (BB)
Type <string of characters> (T(n))
Move cursor to <target coordinates>
    or Point to <target coordinates> (P)
Locate object on screen <object description> (M)
Verify that <description> (M)
Wait for <description> (W(t))
```

**Standard Primitive Mental Operators**

Below follows a brief description of the NGOMSL primitive mental operators; examples of their use appear later.

***Flow of control.*** A submethod is invoked by declaring its goal:

```
Accomplish goal: <goal description>
```

This is analogous to an ordinary CALL statement; control passes to the method for the goal, and returns here when the goal has been accomplished. The operator:

```
Return with goal accomplished
```

is analogous to an ordinary RETURN statement, and marks the end of a method.

A decision is represented by a `Decide` operator; a step may contain only one `Decide` operator, and all other operators in the step have to be contained inside this `Decide`. A `Decide` operator contains either one IF-THEN conditional with an optional ELSE, or any number of IF-THEN conditionals. Here are three examples:

```
1. Decide: If <operator...> Then <operator>

2. Decide: If <operator...> Then <operator>

3. Decide: If <operator...> Then <operator>
           If <operator...> Then <operator>
           If <operator...> Then <operator>
         ...
```

If there are multiple IF-THEN conditionals, as in the third example above, the conditions must be mutually exclusive, so that only one condition can match, and the order in which the If-Thens are listed is irrelevant. The `Decide` operator is for making a simple decision that governs the flow of control within a method. It is not supposed to be used within a selection rule set, which has its own structure (see below). The `IF` clause typically contains operators that test some state of the environment. Notice that the complexity of a `DECIDE` operator is strictly limited; only one simple `ELSE` clause is allowed, and multiple conditionals must be mutually exclusive and independent of order. More complex conditional situations must be handled by separate decision-making methods that have multiple steps, decisions, and branching.

There is a branching operator:

```
Goto Step <number>
```

As in structured programming, a `Goto` is used sparingly; normally it used only with `Decide` operators.

***Memory storage and retrieval.*** The memory operators reflect the distinction between *long-term memory* (LTM) and *working memory* (WM) (often termed *short-term memory*) as they are typically used in computer operation tasks:

```
Recall that <WM-object-description>
Retain that <WM-object-description>
Forget that <WM-object-description>
Retrieve-from-LTM that <LTM-object-description>
```

The terminology is not ideal, but there are few choices in English. `Recall` means to fetch from working memory; `Retain` means to store in working memory, while `Forget` means that the information is no longer needed, and so can be dropped from working memory (although counter-intuitive, this is a real phenomenon; see Bjork, 1972). The methods presented here assume that information is not lost from working memory, so `Forget` refers only to the deliberate dropping of information. Any problems due to "memory overload" could be identified by looking at how much has been retained relative to forgotten (see below, on mental workload). There is only a `Retrieve` operator for LTM, because in the tasks typically modeled, long-term learning and forgetting are not involved. The WM operator execution time is always bundled into the time to execute the step. The `Retrieve-from-LTM` operator can take a **M** time or longer to execute if the user in inexperienced, but with practice, this operator time also becomes bundled into the step execution time.

**Analyst-Defined Mental Operators**

As discussed in some detail below, the analyst will often encounter psychological processes that are too complex to be practical to represent as methods in the GOMS model and that often have little to do with the specifics of the system design. The analyst can bypass these processes by defining operators that act as place holders for the mental activities that will not be further analyzed. Depending on the specific situation, such operators may correspond Keystroke-Level Model **M** operators, and so can be estimated to take 1.2 sec, but some, such as `Make up your mind` below, clearly will take much longer.

Some examples of analyst-defined mental operators:

`Get-from-task <name>` represents the process of accessing or thinking of a task parameter designated by `<name>` and putting the information into working memory.

`Get-next-edit-location` represents the process of scanning a marked up manuscript to determine the location of the next edit.

`Think-of <description>` represents a process of thinking of a value for some parameter designated by `<description>` and putting the information into working memory.

`Read <name> value from screen` represents the process of interpreting characters on a screen that supply a value for some parameter designated by `<name>` and putting the information into working memory.

`Most-recent` represents determining the most recent of two time-stamped items

`Make up your mind <decision description>` represents a complicated decision making process designated by `<decision description>` that puts the final result into working memory.

## 2.3 Methods

A method is a sequence of steps that accomplishes a goal. A step in a method typically consists of an external operator, such a pressing a key, or a set of mental operators involved with setting up and accomplishing a subgoal. Much of the work in analyzing a user interface consists of specifying the actual steps that users carry out in order to accomplish goals, so describing the methods is the focus of the task analysis.

The form for a method is as follows:

```
Method for goal: <goal description>
    Step 1. <operator> ...
    Step 2. <operator> ...
    ...
    Step n. Return with goal accomplished.
```

Note that more than one <operator> can appear in a step (see guidelines below), and that the last step must contain only the operator `Return with goal accomplished.` Methods often call sub-methods to accomplish goals that are subgoals. This method hierarchy takes the following form:

```
Method for goal: <goal description>
    Step 1. <operator>
    Step 2. <operator>
    ...
    Step i. Accomplish goal: <subgoal description>
    ...
    Step m. Return with goal accomplished.

Method for goal: <subgoal description>
    Step 1. <operator>
    Step 2. <operator>
    ...
    Step j. Accomplish goal: <sub-subgoal description>
    ...
    Step n. Return with goal accomplished.
...
```

### 2.4 Selection Rules

The purpose of a *selection rule* is to route control to the appropriate method to accomplish a goal. Clearly, if there is more than one method for a goal, then a selection rule is logically required.

There are many possible ways to represent selection rules. In the approach presented here, a selection rule responds to the combination of a *general* goal and a specific context by setting up a *specific* goal of executing one of the methods that will accomplish the general goal. In other words, a selection rule specifies that for a particular general goal, and certain specific properties of the situation, then the general goal should be accomplished by accomplishing a situation-specific goal.

For example, in a text editor with a find function, if the general goal is to move to a certain place in the text, and the specific context is that the place is visible on the screen, then the general goal should be accomplished by accomplishing the specific goal of `move-with-cursor-keys`. But if the place is far away, then the general goal should be accomplished by the specific one of `move-with-find-function`.

If the analyst discovers that there is more than one method to accomplish a goal, then the general goal should be decomposed into a set of specific goals, one for each method. The analyst should then devise a set of mutually exclusive conditions that describe which method should be used in what contexts.

In the notation introduced here, selection rules are `If-Then` rules that are grouped into sets that are governed by a general goal. If the general goal is present, the conditions of the rules in the set are tested in parallel to choose the specific goal to be accomplished. The relationship with the underlying production rule models is very direct (see Bovair, Kieras, & Polson, 1990). The form for a selection rule set is:

```
Selection rule set for goal: <general goal description>
    If <condition> Then accomplish goal: <specific goal description>.
    If <condition> Then accomplish goal: <specific goal description>.
    ...
Return with goal accomplished.
```

Each `<condition>` consists of one or more operators that test working memory, test contents of the task description, or test the external perceptual situation; these operators cannot be motor operators such as pressing a key. All of the operators in a condition have to be true for the condition to be true. Notice that the `Decide` operator is not used here. The order of the `If-Then` statements is not supposed to be significant; but they need to be written so that only one of the conditions can be true at a time. After the specific goal is accomplished by one of the `If-Then` statements, then the general goal is reported as accomplished.

A simple example for moving the cursor in a text editor:

```
Selection rule set for goal: move the cursor
If destination visible on screen Then
    accomplish goal: moving-with-arrow-keys.
If destination not visible on screen and distance is short, Then
    accomplish goal: moving-with-scroll-keys.
If destination not visible on screen and distance is long and
    the task description contains a find string, Then
    accomplish goal: moving-with-find-function.
Return with goal accomplished
```

The notation for a selection rule set resembles that for a method; it is like a method except for the property that the flow of control through the body of a selection rule set is not sequential, step-by-step, but instead follows whichever `IF` is true, and then continues with the final `Return with goal accomplished` operator. A common and natural confusion is when a `Selection rule set` should be used and when a `Decide` should be used. A `Selection rule set` is used exclusively to route control to the suitable method for a goal, and so can only have `accomplish goal` operators in the THEN clause, while a `Decide` operator controls flow of control within a method, and can have any type of operator in the THEN clause. Thus, if there is more than one method to accomplish a goal, use an `Selection rule set` to dispatch to the more specific method. If you want to control which operators in what sequence are executed within a method, use a `Decide`. Unfortunately, this distinction is still not entirely clear because a `Decide` with multiple If-Thens can masquerade as a `Selection rule set`; further experience will be needed to refine the distinction.

## 2.5 Task Descriptions and Task Instances

### Task Description

A *task description* describes a generic task in terms of the goal to be accomplished, the situation information required to specify the goal, and the auxiliary information required to accomplish the goal that might be involved in bypassing descriptions of complex processes (see below). Thus, the task description is essentially the "parameter list" for the methods that perform the task.

Example: A sample task description for deleting text with a certain word processor contains the following items:

- the goal is to delete a piece of arbitrary text
- the starting location of the text
- the ending location of the text
- a find string for locating the beginning of the text

The goal associated with the task is described, along with the specifics of the task, namely what text is to be deleted. A piece of auxiliary information is the find string for locating the text.

The way to think of a task description is that it is a description of the data that the GOMS model needs in order to carry out the tasks. So the combination of the methods in the GOMS model, and the information in the task description, completely describes the knowledge that the user must have to accomplish the tasks. In the above example, including the find string in the task description means that we are assuming that accomplishing this task efficiently will involve using the find function. But by including the actual find string in the task description, we are making clear that our GOMS model methods are not responsible for computing the find string; we are assuming that users come up with it, but we are choosing not to describe how they do this.

**Task Instance**

A *task instance* is a description of a specific task. It consists of specific values for all of the "parameters" in a task description. For example, a task instance for the above task description would be:

- the goal is to delete a piece of arbitrary text
- the starting location of the text is line 10, column 1
- the ending location of the text is line 11 column 17
- a find string for locating the beginning of the text is "Now is the"

Notice that much of this example is essentially the same information in a mark-up on a paper manuscript.

If one has correctly specified a set of methods in a GOMS model, then one should be able to correctly execute a series of task instances by executing the steps in the methods using the specific values in the task instances.

## 2.6 NGOMSL Statements

**Counting NGOMSL Statements**

The estimation procedures used below involve counting the number of NGOMSL Statements. These are defined as follows:

- The step statement counts as one statement regardless of the number or kind of operators, including a `Return with goal accomplished` operator:
  `Step n. <operator>...`

- The method statement must also be counted as one statement:
  `Method for goal: <goal description>`

- The selection rule set statement counts as one statement:
  `Selection rule set for goal: <general goal description>`

- The IF-THEN statement used in a selection rule counts as one statement:
  `If <condition> Then accomplish goal: <specific goal description>.`

- The terminating statement of a selection rule set counts as one statement:
  `Return with goal accomplished.`

- Steps containing a `Decide` operator involve a special case. As mentioned before, a step may contain only one `Decide` operator, and any other operators must be inside the `Decide` operator.

Each IF-THEN or ELSE corresponds to a production rule, so to determine the number of statements that a `Decide` step should be counted as, count the number of If-Thens or Elses contained in the `Decide` operator; the result is the number of statements for the step. Thus, If the operator consists of a simple IF-THEN operator, such as:

```
Decide: If <operator...> Then <operator>
```

the step counts as one statement. However, if it contains IF-THEN-ELSE:

```
Decide: If <operator...> Then <operator> Else <operator>
```

it counts as two statements. The following multiple-conditional `Decide` has three IF-THENS, and so counts at three statements:

```
Decide: If <operator...> Then <operator>
        If <operator...> Then <operator>
        If <operator...> Then <operator>
```

**Execution of NGOMSL Statements**

The NGOMSL expressions defined as "statements" above are actually executed, and so have to get counted in estimates of execution time. In particular, the `Method` statement that begins a method is counted as being executed. In terms of the underlying production rule model, the `Method` statement corresponds to a production rule that performs some housekeeping functions. (In most computer languages, the PROCEDURE statement actually compiles into some executed housekeeping code.)

Thus, the "overhead" time to call a method always consists of three NGOMSL statement execution times: one for the step in the calling method that contains the `Accomplish goal` operator that calls the method, one for the `Method` statement at the beginning of the called method, and one for the step in the called method that contains the `Return with goal accomplished` operator.

The statements in a selection rule set execute in a special way. The `Selection rule set` statement is executed first. Then, *only one* of the `If-Then` statements is executed, namely the one whose condition is met. In the production rule cognitive architecture, the conditions of all productions are tested simultaneously, in parallel, and only the one whose condition is satisfied is actually executed. Finally, the `Return with goal accomplished` statement is executed last. Thus, the time to execute a selection rule set is always 3 NGOMSL statement execution times; one for the `Selection rule set` statement, one for the `If-Then` that executes, and one for the `Return with goal accomplished`. This is the case regardless of how many `If-Thens` there are in the `Selection rule set`.

Likewise, in steps containing `Decide` operators, count toward execution time only one of the IF-THEN or ELSE clauses, no matter how many there are in the step.

## 3. GENERAL ISSUES IN GOMS TASK ANALYSIS

### 3.1 Judgment Calls

In performing a GOMS task analysis, the analyst is repeatedly making decisions about:

- how users view the task in terms of their natural goals,
- how they decompose the task into subtasks,
- what the natural steps are in the user's methods.

It is possible to collect data on how users view and decompose tasks, but often it is not practical to do so. Thus, in order to do a useful task analysis, the analyst must make judgment calls on these issues. These are decisions based on the analyst's judgment, rather than on systematically collected behavioral data. In making judgment calls, the analyst is actually constructing a psychological theory or model for how people do the task, and so will have to make speculative, hypothetical claims and assumptions about how users think about the task. Because the analyst does not normally have the time or opportunities to collect the data required to test alternative models, these decisions may be wrong, but making them is better than not doing the analysis at all. By documenting these judgment calls, the analyst can explore more than one way of decomposing the task, and consider whether there are serious implications to how these decisions are made. If so, collecting behavioral data might then be required. But notice that once the basic decisions are made for a task, the methods are determined by the design of the system, and no longer by judgments on the part of the analyst.

For example, in the extended example below for moving text in MacWrite, the main judgment call is that due to the command structure, the user views moving text as first cutting, then pasting, rather than as a single unitary move operation. Given this judgment, the actual methods are determined by the possible sequences of actions that MacWrite permits to do cutting and pasting.

In contrast, on the IBM DisplayWriter, the design does not allow you to use the cut and paste operations embedded in the MOVE command separately. So here, the decomposition of moving into "cut then paste" would be a weak judgment call. The most reasonable guess is that a DisplayWriter user thinks of MOVE not in terms of cut and paste subgoals, but in terms of first selecting the text, then issuing Move command, and then designating the target location. So what is superficially the same text editing task may have different decompositions into subgoals, depending on how the system design encourages the user to think about it.

It could be argued that it is inappropriate for the analyst to be making *assumptions* about how humans view a system. However, notice that any designer of a system has in fact made many such assumptions. The usability problems in many software products are a result of the designer making assumptions, often unconsciously, with little or no thoughtful consideration of the implications for users. So, the analyst's assumptions, since they are based on a careful consideration from the user's point of view, can not do any more harm than that typically resulting from the designer's assumptions, and should lead to better results.

### 3.2 Pitfalls in Talking to Users

If the system already exists and has users, the analyst can learn a lot about how users view the task by talking to the users. You can get some ideas about how they decompose the task in to subtasks and what methods and selection rules they use.

However, remember that a basic lesson from the painful history of cognitive psychology is that people have only a very limited awareness of their own goals, strategies, and mental processes in general. Thus the analyst can

not simply collect this information from interviews or having people "think out loud." What users *actually* do can differ a lot from what they *think* they do. The analyst will have to combine information from talking to users with considerations of how the task constrains the user's behavior, and most importantly, observations of actual user behavior. So, rather than ask people to describe verbally what they do, try to arrange a situation where they demonstrate on the system what they do, or better yet, you observe what they normally do in an unobtrusive way.

In addition, what users actually do with a system may not in fact be what they *should* be doing with it. As a result of poor design, bad documentation, or inadequate training, users may not in fact be taking advantage of features of the system that allow them to be more productive. The analyst should try to understand why this is happening, because a good design will only be good if it is used in the intended way. But for purposes of a GOMS analysis, the analyst will have to decide whether to assume a sub-optimal use of the system, or a fully informed one.

### 3.3 Bypassing Complex Processes

Many cognitive processes are too difficult to analyze in a practical context. Examples of such processes are reading, problem-solving, figuring out the best wording for a sentence, finding a bug in a computer program, and so forth. The approach presented here is to bypass the analysis of a complex process by simply representing it with a "dummy" or "placeholder" operator. In this way the analyst does not lose sight of the presence of the process, and can determine many things about what influence it might have on the user's performance with a design. Representing a bypassed process consists of using an analyst-defined operator together with information in the task description as place holders to document that the process is taking place and what its assumed results are.

For example, in MacWrite, the user may use tabs. How does the user know, or figure out, where to put them? The analyst might assume that the difficulties of doing this have nothing to do with the design of MacWrite (which may or not be true). The analyst can bypass the process of how the user figures out tab locations by assuming that user has figured them out already, and includes the tab settings as part of the task description supplied to the methods. The analyst defines a special operator that is used by the methods to access this information when it is needed (cf. the discussion in Bennett, Lorch, Kieras, & Polson, 1987).

As a second example, how does the user know that a particular scribble on the paper means "delete this word?" The analyst can bypass this problem by putting in the task description the information that the goal is to `Delete` and that the target text is at such-and-such a location (see example task descriptions above), and then using an analyst-defined operator that accesses the task description, such as `Look-at-document`. The methods will invoke this operator at the places where the user is assumed to have to look at the document to find out what to do. This way, the contents of the task description show the results of the complex reading process that was bypassed, and the places in the methods where the operator appears mark where the user is engaging in the complex reading process.

The analyst should only bypass processes for which a full analysis would be irrelevant to the design. But sometimes the complexity of the bypassed process is related to the design. For example, a text editor user must be able to read the paper marked-up form of a document, regardless of the design of the text editor, meaning that the reading process can be bypassed because it does not need to be analyzed in order to choose between two different text editor designs. On the other hand, the POET editor (see Card, Moran, & Newell, 1983) requires heavy use of find-strings which the user has to devise as needed. This process can still be bypassed with an analyst-defined operator, `think-up-find-string`, and the actual find strings specified in the task description. But suppose we are comparing POET to an editor that does not require such heavy use of find strings. Any conclusions about the difficulty of POET compared to the other editor will depend critically how hard the `think-up-find-string` operator is to execute. Thus, bypassing a process might produce seriously misleading results.

**Representing Bypassed Processes**

A bypassed process could be represented just by defining a complex mental operator and using it wherever it is needed in the methods. For example, we could define a `Most-Recent` operator that determines which of two time-stamped items (e.g. files) is the most recent. Exactly how the human cognitive process is done is irrelevant to the analysis. Once we have finished the analysis, we can, however, determine how often this operator is used, and thus see how important it is. Some designs may differ in how often complex processes are executed; if so, we can estimate or measure the time involved for the operators, and thus determine whether our design decisions are sensitive to these estimates.

But it seems to be better to put the results of a complex mental process into the task description, and then define a simple operator that just accesses this result when it is needed. This can be called the "yellow pad" heuristic; the concept is that the user has a complete description of the task written out on a yellow pad; all of the "thinking" unconnected with interacting with the system has already been done. The user's task is then only to interact with the system in order to carry out this completely described task, referring to the yellow pad as necessary to obtain the required information. Notice that this approach was followed in Card, Moran, and Newell's (1983, Ch.5) analysis of text-editing; their user is assumed to be working from a marked-up manuscript of the edited document, rather than composing the document or figuring out the desired changes while working at the computer. A similar approach was used in the Bennett, Lorch, Kieras, and Polson (1987) analysis of the task of entering a complex data table using different document preparation systems.

The reason why the yellow-pad approach is desirable is that the judgment calls on when and how much such complex processing is done seem to be relatively inconsistent, primarily because it is not very constrained by the design of the user interface. For example, one analyst might think that it takes 3 episodes of complex thinking to arrive at where to set a tab position in preparing a document, while another might think it would take 5, even though in both cases the methods involved in actually setting the location would be the same. Now, each such mental activity could be represented with a bypassing operator, which would normally be assigned a relatively large execution time estimate (e.g. at least 1.2 sec). But, since most other operators, like keystrokes, take a much shorter time, the uncertainty in the number of complex mental operators will cause the estimate of execution time to vary drastically, probably washing out the other differences between two similar designs. This weakening of the analysis is unnecessary, since these mental processes are often unrelated to the complexity of the methods entailed by the interface design.

The yellow-pad heuristic, by moving the complex mental processes "off-line" with just their results in the task description, eliminates these complex processes from the methods themselves. Thus the estimates of execution time depend much more heavily on the methods entailed by the actual interface design. In the meantime, the complex processes have not been ignored; their presence is documented in the task description. As discussed more below, at a later time, the analyst can consider whether these bypassed processes play an important role in the usability of the design, especially if two designs differ substantially in which processes are involved.

## 3.4 Analyze a General Set of Tasks, Not Specific Instances

Often, user interface designers will work with *task scenarios*, which are essentially descriptions in ordinary language of task instances and what the user would do in each one. The list of specific actions that the user would perform for a specific task can be called a *trace*, analogous to the specific sequence of results one obtains when "tracing" a computer program. Assembling a set of scenarios and traces is often useful as an informal way of characterizing a proposed user interface and its impact on the user.

If one has collected a set of task scenarios and traces, the natural temptation is to construct a description of the user's methods for executing these specific task instances. This temptation must be resisted; the goal of GOMS task

analysis is a description of the *general* methods for accomplishing a set of tasks, not just the method for executing a specific instance of a task.

If you fall into the trap of writing methods for specific task instances, chances are that you will describe methods that are "flat," containing little in the way of method and submethod hierarchies, and which also may contain only specific keystroke operations. E.g., if the task scenario is that the user deletes the file FOOBAR, such a method will generate the keystroke sequence of "DELETE FOOBAR <CR>." But the fatal problem is that a tiny change in the task instance means that the method will not work. What if the task is to delete the file "FOO?" Sorry, you don't have a method for that! This corresponds to a user who has memorized by rote how to do an exact task, but who can't execute variations of the task.

On the other hand, a set of *general* methods will have the property that the information in a specific task instance acts like "parameters" for a general program, and the general methods will thus generate the specific actions required to carry out that task instance. Any task instance of the general type will be successfully executed by the general method. For example, a general method for deleting the file specified by <filename> will generate the keystroke sequence of "DELETE " followed by the string <filename> followed by <CR>. This corresponds to a user who knows how to use the system in the general way normally intended.

So, what should you do with a collection of task scenarios or traces? Study them to discover the range of things that the user has to do. Then set them aside and write a set of general methods, using the approach described below, that can correctly perform any specific task within the classes defined by your methods (e.g., delete any file whose name is specified in the task description). You can check to see if the methods will generate the correct trace for each task scenario, but they should also work for *any* scenario of the same type.

### 3.5 When Can a GOMS Analysis be Done?

**After Implementation - Existing Systems**

Constructing a GOMS model for a system that already exists is the easiest case for the analyst because much of the information needed for the GOMS analysis can be obtained from the system itself, its documentation, its designers, and the present users. The user's goals can be determined by considering the actual and intended use of the system; the methods are determined by what actual steps have to be carried out. The analyst's main problem will be to determine whether what users actually do is what the designers intended them to do, and then go on to decide what the users' actual goals and methods are. For example, the documentation for a sophisticated document preparation system gave no clue to the fact that most users dealt with the complex control language by keeping "template" files on hand which they just modified as needed for specific documents. Likewise, this mode of use was apparently not intended by the designers. So the first task for the analyst is to determine how an existing system is actually used in terms of the goals that actual users are trying to accomplish. Talking to, and observing, users can help the analyst with these basic decisions (but remember the pitfalls discussed above).

Since in this case the system exists, it is possible to collect data on the user's learning and performance with the system, so using a GOMS model to predict this data would only be of interest if the analyst wanted to verify that the model was accurate, perhaps in conjunction with evaluating the effect of proposed changes to the system. However, notice that collecting systematic learning and performance data for a complex piece of software can be an extremely expensive undertaking; if one is confident of the model, it could be used as a substitute for empirical data in activities such as comparing two competing existing products.

**After Design Specification - Evaluation During Development**

There is no need for the system to be already implemented or in use for a GOMS analysis to be carried out. It is only necessary that the analyst can specify the components of the GOMS model. If the design has been specified in adequate detail, then the analyst can identify the intended user's goals and describe the corresponding methods just as in the case of an existing system.

Of course, the analyst can not get the user's perspective since there are as yet no users to talk to. However, the analyst can talk to the designers to determine the designer's intentions and assumptions about the user's goals and methods, and then construct the corresponding GOMS model as a way to make these assumptions explicit and to explore their implications. Predictions can then be made of learning and performance characteristics, and then used to help correct and revise the design. The analyst thus plays the role of the future user's advocate, by systematically assessing how the design will affect future users. Since the analysis can be done before the system is implemented, it should be possible to identify and put into place an improved design without wasting coding effort.

However, the analyst can often be in a difficult position. Even fairly detailed design specifications often omit many specific details that directly affect the methods that users will have to learn. For example, the design specifications for a system may define the general pattern of interaction by specifying pop-up menus, but not the specific menu choices available, or which choices users will have to make to accomplish actual tasks. Often these detailed design decisions are left up to whoever happens to write the relevant code. The analyst may not be able to provide many predictions until the design is more fully fleshed out, and may have to urge the designers to do more complete specification than they normally would.

**During Design - GOMS Analysis Guiding the Design**

Rather than analyze an existing or specified design, the interface could be designed concurrently with describing the GOMS model. That is, by starting with listing the user's top-level goals, then defining the top-level methods for these goals, and then going on to the subgoals and submethods, one is in a position to make decisions about the design of the user interface directly in the context of what the impact is on the user. For example, bad design choices may be immediately revealed as spawning inconsistent, complex methods, leading the designer quickly into considering better alternatives. Clearly, this approach is possible only if the designer and analyst are closely cooperating, or are the same person.

Perhaps counter to intuition, there is little difference in the approach to GOMS analysis between doing it *during* the design process and doing it after. Doing the analysis during the design means that the analyst and designer are making design decisions about what the goals and methods *should be*, and then immediately describing them in the GOMS model. Doing the analysis *after* the system is designed means that the analyst is trying to determine the design decisions that were made *sometime in the past*, and then describing them in a GOMS model. For example, instead of determining and describing how the user does a cut-and-paste with an existing text editor, the designer-analyst *decides* and describes how the user *will* do it. It seems clear that the reliability of the analysis would be better if it is done during the design process, but the overall logic is the same in both cases.

## 4. A PROCEDURE FOR CONSTRUCTING A GOMS MODEL

The analysis of a task is done top-down from the most general user goal to more specific subgoals, with primitive operators finally at the bottom. All of the goals at each level are dealt with before going down to a lower level. The recipe presented here is based on this idea of thinking in terms of a top-down, breadth-first expansion of methods.

In overview, you start by describing a method for accomplishing a top-level goal in terms of high-level operators. Then you provide methods for performing the high-level operators in terms of lower-level operators. Then provide methods for these operators, and continue until you have arrived at enough detail to suit your needs, or until the methods are expressed in terms of primitive operators. So, as the analysis proceeds, high-level operators are replaced by goals to be accomplished by methods that involve lower-level operators. When you provide a method for a high-level operator, performing the operator becomes a goal, and so you provide a method for accomplishing that goal.

It is important to perform the analysis breadth-first, rather than depth-first. By considering all of the methods that are at the same level of the hierarchy before getting more specific, you are more likely to notice how the methods are similar to each other; such method similarities are critical to capturing the "consistency" of the user interface (see below).

You can choose to analyze in detail only selected portions of the user interface, and can leave at the level of high-level operators those portions where detail is not needed, or not possible. For example, suppose we aren't concerned with the specific keystrokes required to start a mail program that runs on different time-sharing systems, but are concerned with how to operate the mail program itself. We might describe the method to check for mail as follows:

```
Method for goal: check mail
    Step 1. LOG-INTO-SYSTEM
    Step 2. START-MAIL-PROGRAM
    Step 3. TYPE-IN "RETRIEVE<CR>"
    ... etc.
```

Logging in and starting the mail program are described with high-level operators, but we get down to specific keystrokes (the user types "RETRIEVE<CR>") once we are dealing with the mail program. If we are only concerned with the user interface of the mail program, we may choose to leave the high-level operators in Step 1 and Step 2 as unanalyzed. If so, we have chosen to bypass these processes, and are representing them with simple placeholders. We would then go on to describe in detail the methods involved in dealing with the mail program.

### 4.1 Summary of Procedure

Hint: since this is like writing and revising a computer program, use a text editor to allow fast writing and modification. An outline processor works especially well.

Step A: Choose the top-level user's goals

Step B: Do the following recursive procedure:

B1. Draft a method to accomplish each goal

B2. After completing the draft, check and rewrite as needed for consistency and conformance to guidelines.

B3. If needed, go to a lower level of analysis by changing the high-level operators to accomplish-goal operators, and then provide methods for the corresponding goals.

Step C: Document and check the analysis.

Step D: Check sensitivity to judgment calls and assumptions.

### 4.2 Detailed Description of Procedure

**Step A: Choose the top-level user's goals**

The top-level user's goals are the first goals that you will expand upon in the top-down analysis.

***Advantages of starting with high-level goals.*** It is probably worthwhile to make the top-level goals very high-level, rather than lower-level, to capture any important relationships within the set of tasks that the system is supposed to address. An example for a text editor is that a high level goal would be `revise document`, while a lower-level one would be `delete text`. Starting with a set of goals at too low a level entails a risk of missing the methods involved in going from one type of task to another.

For example, many Macintosh applications combine deleting and inserting text in an especially convenient way. The goal of `change word` has a method of its own; i.e., double click on the word and then type the new word. If you start with `revise document` you might see that one kind of revision is changing one piece of text to another, and so you would consider the corresponding methods. If you start with goals like `insert text` and `delete text` you have already decided that this is the breakdown into subgoals, and so are more like to miss a case where the user has a natural goal that cuts across the usual functions.

As an example of very high-level goals, consider the goal of `produce document` in the sense of ”publishing” - getting a document actually distributed to other people. This will involve first creating it, then revising it, and then getting the final printed version of it. In an environment that includes a mixture of ordinary and desktop publishing facilities, there may be some important subtasks that have to be done in going from one to the other of the major tasks, such as taking a document out of an ordinary text editor and loading it into a page-layout editor, or combining the results of a text and a graphics editor. If you are designing just one of these packages, say the page-layout editor, and start only with goals that correspond to page-layout functions, you may miss what the user has to do to integrate the use of the page-layout editor in the rest of the environment.

***Most tasks have a unit-task control structure.*** Unless you have reason to believe otherwise, assume that the task you are analyzing has a unit-task type of control structure. This means that the user will accomplish the overall task by doing a series of smaller tasks one after the other. For a system such as a text editor, this means that the top-level goal of `edit document` will be accomplished by a unit-task method similar to that described by Card, Moran, and Newell, (1983). One way to describe this type of method in NGOMSL is as follows:

```
Method for goal: edit the document
    Step 1. Get next unit task information from marked-up manuscript.
    Step 2. Decide: If no more unit tasks, then return with goal accomplished.
    Step 3. Accomplish goal: move to the unit task location.
    Step 4. Accomplish goal: perform the unit task.
    Step 5. Goto 1.
```

The goal of performing the unit task typically is accomplished via a selection rule set, which dispatches control to the appropriate method for the unit task type, such as:

```
Selection rule set for goal: perform the unit task
    If the task is deletion, then
           accomplish goal: perform deletion procedure.
    If the task is copying, then
           accomplish goal: perform copy procedure.
    ... etc. ...
    Return with goal accomplished.
```

This type of control structure is common enough that the above method and selection rule set can be used as a template for getting the NGOMSL started. The remaining methods in the analysis will then consist of the specific methods for these subgoals, similar to those described in the extended example below.

**Step B. Do the Following Recursive Procedure:**

**Step B1. Draft a Method to Accomplish Each Goal**

Simply list the series of steps the user has to do. Each step should be a single natural unit of activity. Heuristically, this is just an answer to the question "how would a user describe how to do this?"

Make the steps as general and high-level as possible for the current level of analysis. A heuristic is to consider how a user would describe it in response to the instruction "don't tell me the details yet."

Use the principles for the Keystroke-Level Model for guidance on certain sequence of steps involving mental operators, such as locates before points.

Define new high-level operators, and bypass complex psychological processes as needed. Make a note of the new operators and task description information.

Make simplifying assumptions as needed, such as deferring the consideration of possible shortcuts that experienced users might use. Make a note of these assumptions in comments in the method.

If there is more than one method for accomplishing the goal, draft each method and then draft the selection rule set for the goal. My recommendation is to make the simplifying assumption that alternative methods are not used, and defer consideration of minor alternative methods until later. This is especially helpful for alternative "shortcut" methods.

**Some Guidelines for Step B1**

***How specific?*** As a rule of thumb, you should probably not be describing specific keystroke sequences until about four or so levels down, for typical top-level goals. For example:

```
goal of editing the document
   goal of copying text
      goal of selecting the text
         PRESS SELECT KEY
```

If the draft method involves keystroke sequences sooner, there is probably more structure to the user's goals than the draft method is capturing. Look for similarities between how different goals are accomplished; for example, many editors use a common selection method, suggesting that the user will have this as a subgoal, as in the above example. Really unusual or very poor designs may be exceptions.

***How many steps in a method?*** As another rule of thumb, if there are more than 5 or so steps in the method, it may not be at right level of detail; the operators used in the steps may not be high-level enough. See if a sequence of steps can be made into a high-level operator, especially if the same sequence appears in other methods. Describing the methods breadth-first should help with noticing such shared sequences.

Example: Too many steps because level of detail is too fine:

```
Method for goal: start edit session
    Step 1. Type-in "edit"
    Step 2. Type-in filename
    Step 3. Press-key CR
    Step 4. If main menu present, Press-Key 1
    Step 5. Press-key CR
    etc.
```

Probably there should be higher-level operators:

```
Method for goal: start edit session
    Step 1. Enter editor with filename
    Step 2. Choose revise mode
    etc.
```

As in the above guideline, if there are too many steps, it may be due to a bad design, or there may be more structure to the user's knowledge than you are assuming.

***How many operators in a step?*** How many operators can be done in a single cognitive step is an important question in the fundamental, and still developing, theory of cognitive skill (cf. Anderson's composition concept, Anderson, 1982). Based on Bovair, Kieras, and Polson (1990), these guidelines are reasonable:

- Use no more than one accomplish-goal operator to a step.

- Use no more than one high-level operator to a step.

- For ordinary or novice users, use no more than one external primitive operator to a step.

- For expert, very well practiced users, several external primitive operators can be used in a single step as follows: If there is a sequence of external primitive operators that is often performed without any decisions or subgoals involved, then this sequence of operators can appear in a single step.

***Some standard mental operator sequences.*** Certain operators should be included in a procedure; these guidelines are similar to those associated with the Keystroke Level Model for the placement of mental operators (Card, Moran, & Newell, 1983). There should be a `Locate` operator executed prior to a `Point`, to reflect that before an object can be pointed to, its location must be known. If the system provides feedback to the user, then there should be a `Verify` operator in the method to represent how the user is expected to notice and make use of that feedback information. A `Verify` operator should normally be included at the point were the user must commit to an entry of information, such as prior to hitting the Return key in a command line interface. Finally, there should be a mental operator such as `Think-of` or `Get-from-task` for obtaining each item of task parameter information.

***Developing the task description.*** A good way to keep from being overwhelmed by the details involved in describing a task is to develop the task description in parallel with the methods. That is, put in the task description only what is needed for the methods at the current level of the analysis. As the analysis deepens, add more detail and precision to the task description.

For example, early in the description of text editing methods, the type of edit (delete, move, etc.) may be needed by a method. But not until later, when you are describing the very detailed methods, will you need to specify information such as the exact position of the edit, what will be moved or deleted, and other detailed information.

**Step B2. Check for Consistency and Conformance to Guidelines**

• Check on the level of detail and length of each method.

• Check that you have made consistent assumptions about user's expertise with regard to the number of operators in a step.

• Identify the high-level operators you used; check that each high-level operator corresponds to a natural goal; redefine the operator or rewrite the method if not so.

• Maintain a list of operators used in the analysis, showing which methods each operator appears in.
Check for consistency of terminology and usage with already defined operators. For example, we could end up with:

```
Look-for <manuscript markup information>
Look-at-manuscript-markup <information>
Look <markup-type>
```

when we probably should have just:

```
Look-at-manuscript-markup <type>
```

Redefine new or old operators to ensure consistency; and add new operators to the list.

• Examine any simplifying assumptions made, and elaborate the method if useful to do so.

**Step B3. If Needed, Go to the Next Lower Level of Analysis**

If all of the operators in a method are primitives, then this is the final level of analysis of the method, and nothing further needs to be done with this method. If some of the operators are high-level, non-primitive operators, examine each one and decide whether to provide a method for performing it. The basis for your decision is whether additional detail is needed for design purposes. For example, early in the design of a word processing system, it might not be decided whether the system will have a mouse or cursor keys. Thus it will not be possible to describe cursor movement and object selection below the level of high-level operators. In general, you should plan to expand as many high-level operators as possible into primitives at the level of keystrokes, because many important design problems, such as a lack of consistent methods, will show up mainly at this level of detail. Also, the time estimates are clearest and most meaningful at this level. If you choose to provide a method for an operator, rewrite that step in the method (and in all other methods using the operator). Replace the operator with an accomplish-goal operator for the corresponding goal. Update the operator list, and apply this recipe to describe the method for accomplishing the new goal.

For example, suppose the current method for copying text is:

```
Method for goal: copy text
    Step 1. Select the text.
    Step 2. Issue COPY command.
    Step 3. Return with goal accomplished.
```

and we choose to provide a method for the Step 1 operator `Select the text.` We rewrite the copying text method as:

```
Method for goal: copy text
>   Step 1. Accomplish goal: select the text.
    Step 2. Issue COPY command.
    Step 3. Return with goal accomplished.
```

and then provide a method for the goal of selecting the text. To make the example clear, the changed line is shown in **Boldface** with a **>** in the left margin marking the changed line.

**Step C. Documenting and Checking the Analysis**

After you have completed writing out the NGOMSL model, list the following items of documentation:

- Primitive external operators used
- Analyst-defined operators used, along with a brief description of each one
- Assumptions and judgment calls that you made during the analysis
- The contents of the task description for each type of task

Then, choose some representative task instances, and check on the accuracy of the model by executing the methods as carefully as possible using hand simulation, and noting the actions generated by the model. This can be tedious. Verify that the sequences of actions are actually correct ways to execute the tasks (e.g. try them on the system). For a large project, you may want to consider implementing the methods as a computer program to automate this process. If the methods do not generate correct action sequences, make corrections so that the methods will correctly execute the task instances.

**Step D. Check Sensitivity to Judgment Calls and Assumptions**

Examine the judgment calls and assumptions made during the analysis to determine whether the conclusions about design quality and the performance estimates would change radically if the judgments or assumptions were made differently. This sensitivity analysis will be very important if two designs are being compared that involved different judgments or assumptions; less important if these were the same in the two designs. The analyst may want to develop alternate GOMS models to capture the effects of different judgment calls to systematically evaluate whether they have important impacts on the design.

### 4.3 Making WM Usage Explicit

Explicitly representing how the user must make use of Working Memory (WM) is a way to identify where the system is imposing a high memory load on the user. Probably, representing the WM usage can wait until the rest of analysis is complete; then the NGOMSL methods can be rewritten as needed to make WM usage explicit.

Notice that some kinds of memory are built-in to a GOMS model, such as the goal stack implied by the nesting of methods, and the pointer to the step in a method that is next to be executed. It is not clear theoretically whether this specialized control information is kept in the ordinary WM. In contrast, ordinary WM is definitely used to maintain temporary information that is not a fixed part of a method, such as the name of a file, the location of a piece of text, and so forth. It seems to be rare for ordinary methods on typical pieces of software to exceed WM capacity (cf. Bovair, Kieras, & Polson, 1990); if they did, they probably would have been changed! However, many computer system designs will strain WM in certain situations; often these are the ones where the user will be tempted to write something down on paper.

Representing WM usage requires making the methods explicit on when information is put into WM, accessed, and removed. To represent WM usage in the methods, identify the information that is needed from one step in a

method to another, and describe each item of information in a consistent way. Ensure that when information is first acquired, a `Retain` operator is used, and that each step that needs the information `Recalls` it and when the information is no longer needed, then `Forget` it. . Because the underlying production rule models access to WM during condition matching and rule firing, the times for these basic WM operators are "bundled into the step execution time. Thus, WM operators would normally not occupy a step by themselves, but would always be included in a step that does other operators. For example:

```
Step 3. Find-menu-item "CUT" and retain item position.
Step 4. Recall item position and Move-mouse-to item-position.
Step 5. Forget item position, and press mouse button.
```

A more complete example appears in the extended example that follows.

### 4.4 An Example of Using the Procedure

This example shows the use of NGOMSL notation and illustrates how to construct a GOMS model using the top-down approach. The example system is MacWrite, and the example task is, of course, text editing. Only one type of text editing task, moving a piece of text from one place to another, is analyzed fully. The example consists of a series of passes over the methods, each pass corresponding to a deeper level of analysis. Four passes are shown in this example, but Pass 1 just consists of assuming that the unit task method is used for the topmost user's goal. After Pass 4, the operators, task description, and assumptions are listed.

In each pass, the complete GOMS model is shown. To make it easier to see what is new in each pass, the new material is shown in **boldface**, with the symbol > in the left margin on each new line. Following this example is an illustration of how some of the methods in the example would be modified to make WM use explicit.

**Pass 1**

Our topmost user's goal is editing the document. Taking the above recommendation, we simply start with the unit-task method and the selection rule set that dispatches control to the appropriate method:

```
>Method for goal:edit the document
>   Step 1. Get next unit task information from marked-up manuscript.
>   Step 2. Decide: If no more unit tasks, then return with goal accomplished.
>   Step 3. Accomplish goal:move to the unit task location.
>   Step 4. Accomplish goal:perform the unit task.
>   Step 5. Goto 1.

>Selection rule set for goal:perform the unit task
>   If the task is moving text, then
>          accomplish goal:move text.
>   If the task is deletion, then
>          accomplish goal:delete text.
>   If the task is copying, then
>          accomplish goal:copy text.
>   ... etc. ...
>   Return with goal accomplished.
```

We also need a method for moving to the unit task location. In this example, we don't want to get into the gory details of how scrolling of the display is done. For this reason, the high-level operators in this method will not be further expanded. This method also provides an example of accessing the task description (the marked-up manuscript).

```
>Method for goal: move to the unit task location
>   Step 1. Get location of unit task from manuscript.
>   Step 2. Decide: If unit task location on screen, then
>           return with goal accomplished.
>   Step 3. Use scroll bar to advance text.
>   Step 4. Goto 2.
```

In this example, we will next go on to provide a detailed method just for the goal of moving text.

**Pass 2**

Now, we begin the recursive procedure. Our current top-level goal is moving text. Our first judgment call is assuming that users view moving text as first cutting, then pasting. We write the new method accordingly:

```
Method for goal: edit the document
    Step 1. Get next unit task information from marked-up manuscript.
    Step 2. Decide: If no more unit tasks, then return with goal accomplished.
    Step 3. Accomplish goal: move to the unit task location.
    Step 4. Accomplish goal: perform the unit task.
    Step 5. Goto 1.

Selection rule set for goal: perform the unit task
    If the task is moving text, then
           accomplish goal: move text.
    If the task is deletion, then
           accomplish goal: delete text.
    If the task is copying, then
           accomplish goal: copy text.
    ... etc. ...
    Return with goal accomplished.

Method for goal: move to the unit task location
    Step 1. Get location of unit task from manuscript.
    Step 2. Decide: If unit task location on screen, return with goal
           accomplished.
    Step 3. Use scroll bar to advance text.
    Step 4. Goto 2.

>Method for goal: move text
>   Step 1. Cut text
>   Step 2. Paste text
>   Step 3. Verify correct text moved.
>   Step 4. Return with goal accomplished.
```

Step 1 and Step 2 of the new method are represented here *temporarily* with high-level operators. In the next pass, methods will be provided for them, and the high-level operators will be replaced with accomplish goal operators. Notice that in Step 3 we are assuming that the user will pause to verify that the desired results have been obtained. We will assume (perhaps wrongly) that a similar verification is *not* done within the cutting and pasting methods to be described below.

**Pass 3**

We now provide methods for cutting and pasting. Notice below how steps 2 and 3 of the moving text method have been changed from the previous pass. As an example, the first draft of the method for cutting is too long (see

guidelines); in response to the guideline advice, this is fixed in the second draft.

```
Method for goal: edit the document
    Step 1. Get next unit task information from marked-up manuscript.
    Step 2. Decide: If no more unit tasks, then return with goal accomplished.
    Step 3. Accomplish goal: move to the unit task location.
    Step 4. Accomplish goal: perform the unit task.
    Step 5. Goto 1.

Selection rule set for goal: perform the unit task
    If the task is moving text, then
           accomplish goal: move text.
    If the task is deletion, then
           accomplish goal: delete text.
    If the task is copying, then
           accomplish goal: copy text.
    ... etc. ...
    Return with goal accomplished.

Method for goal: move to the unit task location
    Step 1. Get location of unit task from manuscript.
    Step 2. Decide: If unit task location on screen, return with goal
           accomplished.
    Step 3. Use scroll bar to advance text.
    Step 4. Goto 2.

Method for goal: move text
>   Step 1. Accomplish goal: cut text
>   Step 2. Accomplish goal: paste text
    Step 3. Verify correct text moved.
    Step 4. Return with goal accomplished.

>Method for goal: cut text - First Draft
>   Step 1. Move cursor to beginning of text.
>   Step 2. Hold down mouse button.
>   Step 3. Move cursor to end of text.
>   Step 4. Release mouse button.
>   Step 5. Move cursor to EDIT menu bar item.
>   Step 6. Hold down mouse button.
>   Step 7. Move cursor to CUT item
>   Step 8. Release cursor button.
>   Step 9. Return with goal accomplished
```

Notice that this new method is correctly described, but it has too many steps. Also, this is only the second level of goals, and the method already has external primitive operators. Notice that Steps 1-4 correspond to a general method for how things are selected almost everywhere on the Macintosh, and Steps 5-8 are involved with issuing the CUT command. Perhaps the analysis has stumbled close to providing a trace-based method for executing a specific task rather than general methods that cover the tasks of interest, as discussed above. The second draft of the method corrects the problems with the judgment calls that (1) users know and take advantage of the general selecting function, and so they will have a "subroutine" method for selecting text, and that (2) similarly, they also have a general method for issuing commands. The corresponding sequences in the first draft can be collapsed into two high-level operators, as shown in second draft below of the cutting method. The pasting method is then written in a similar way.

```
>Method for goal: cut text - Second Draft
>   Step 1. Select text.
>   Step 2. Issue CUT command.
>   Step 3. Return with goal accomplished.

>Method for goal: paste text
>   Step 1. Select insertion point.
>   Step 2. Issue PASTE command.
>   Step 3. Return with goal accomplished.
```

**Pass 4**

We now provide some methods for selecting text and the corresponding selection rules, since MacWrite provides several ways of doing this. We also provide methods for selecting the insertion point and issuing CUT and PASTE commands. We make the simplifying assumption that our user does not make use of the command-key shortcuts.

```
Method for goal: edit the document
    Step 1. Get next unit task information from marked-up manuscript.
    Step 2. Decide: If no more unit tasks, then return with goal accomplished.
    Step 3. Accomplish goal: move to the unit task location.
    Step 4. Accomplish goal: perform the unit task.
    Step 5. Goto 1.

Selection rule set for goal: perform the unit task
    If the task is moving text, then
           accomplish goal: move text.
    ... etc. ...
    Return with goal accomplished.

Method for goal: move to the unit task location
    Step 1. Get location of unit task from manuscript.
    Step 2. Decide: If unit task location on screen, return with goal
           accomplished.
    Step 3. Use scroll bar to advance text.
    Step 4. Goto 2.

Method for goal: move text
    Step 1. Accomplish goal: cut text
    Step 2. Accomplish goal: paste text
    Step 3. Verify correct text moved.
    Step 4. Return with goal accomplished.

Method for goal: cut text
>   Step 1. Accomplish goal: select text.
>   Step 2. Accomplish goal: issue CUT command.
    Step 3. Return with goal accomplished.

Method for goal: paste text
>   Step 1. Accomplish goal: select insertion point.
>   Step 2. Accomplish goal: issue PASTE command.
    Step 3. Return with goal accomplished.
```

```
>Selection rule set for goal: select text
>   If text-is word, then
>          accomplish goal: select word.
>   If text-is arbitrary, then
>          accomplish goal: select arbitrary text.
>Return with goal accomplished.

>Method for goal: select word
>   Step 1. Locate middle of word.
>   Step 2. Move cursor to middle of word.
>   Step 3. Double-click mouse button.
>   Step 4. Verify that correct text is selected
>   Step 5. Return with goal accomplished.

>Method for goal: select arbitrary text
>   Step 1. Locate beginning of text.
>   Step 2. Move cursor to beginning of text.
>   Step 3. Press mouse button down.
>   Step 4. Locate end of text.
>   Step 5. Move cursor to end of text.
>   Step 6. Verify that correct text is selected.
>   Step 7. Release mouse button.
>   Step 8. Return with goal accomplished.
```

The last method above seems to be too long. This is the result of a judgment call that the user has to look at the marked-up manuscript to see where the text starts and then find this spot on the screen (Step 1), and then as a separate unit of activity, move the cursor there (Step 2). A similar situation appears in Steps 4 and 5. Some alternative judgment calls: Perhaps there is a drag operator and using it requires determining the end of the text before pressing down the mouse button. Alternately, perhaps the sequence appearing in Steps 1 and 2 and Steps 4 and 5 corresponds to a natural goal of find a place and put the cursor there, for which there should be a high-level operator and later a method. For brevity, these alternative judgment calls are not pursued in this example. The remaining methods are as follows:

```
>Method for goal: select insertion point
>   Step 1. Locate insertion point.
>   Step 2. Move cursor to insertion point.
>   Step 3. Click mouse button.
>   Step 4. Return with goal accomplished.

>Method for goal: issue CUT command
>(assuming that user does not use command-X shortcut)
>   Step 1. Locate "Edit" on Menu Bar
>   Step 2. Move cursor to "Edit" on Menu Bar
>   Step 3. Press mouse button down.
>   Step 4. Verify that menu appears.
>   Step 5. Locate "CUT" in menu.
>   Step 6. Move cursor to "CUT".
>   Step 7. Verify that CUT is selected.
>   Step 8. Release mouse button.
>   Step 9. Return with goal accomplished.

>Method for goal: issue PASTE command
>(assuming that user does not use command-V shortcut)
>   Step 1. Locate "Edit" on Menu Bar
>   Step 2. Move cursor to "Edit" on Menu Bar
```

```
>   Step 3. Press mouse button down.
>   Step 4. Verify that menu appears.
>   Step 5. Locate "PASTE" on menu
>   Step 6. Move cursor to "PASTE".
>   Step 7. Verify that PASTE is selected.
>   Step 8. Release mouse button.
>   Step 9. Return with goal accomplished.
```

**Modifications to Show WM Usage**

To illustrate how WM usage is made explicit, we will rewrite some of the methods to use a generic submethod for issuing a command that captures some of the consistency of the Macintosh menu-based command interface. This generic submethod uses WM to pass a "parameter" that is the name of the command to be issued. The use of working memory is explicitly represented in this method both informally and with variables.

Instead of separate methods for issuing a CUT and a PASTE command, there is now going to be just one method for issuing a command, whose name has been deposited previously in WM. First, the methods that called the previous command-issuing methods need to be modified to put the command name in WM, and to accomplish the generic goal of issuing a command.

```
Method for goal: cut text
    Step 1. Accomplish goal: select text.
>   Step 2. Retain that the command is CUT, and accomplish goal: issue a
            command.
    Step 3. Return with goal accomplished.

Method for goal: paste text
    Step 1. Accomplish goal: select insertion point.
>   Step 2. Retain that the command is PASTE, and accomplish goal: issue a
            command.
    Step 3. Return with goal accomplished.
```

The following method is the generic command-issuing method to replace the previous specialized methods for issuing a CUT and a PASTE command. The command name is like a subroutine parameter, and is passed in through WM. We assume that user must remember with a retrieval from LTM where in the menus the command is, and the user has to remember this menu name while executing the method.

```
Method for goal: issue a command
    Step 1. Recall command name and retrieve from LTM the menu name for it.
    Step 2. Recall menu name, and locate it on Menu Bar.
    Step 3. Move cursor to menu name location.
    Step 4. Press mouse button down.
    Step 5. Verify that menu appears.
    Step 6. Recall command name, and locate it in menu.
    Step 7. Move cursor to command name location.
    Step 8. Recall command name, and verify that it is selected.
    Step 9. Release mouse button.
    Step 10. Forget menu name, forget command name, and return with goal
            accomplished.
```

Notice how this analysis makes explicit that the user has to maintain two chunks in WM during this method; this might be a problem if the higher-level methods required keeping track of 3 - 5 more chunks.

**Final GOMS Model**

Below is the final version of the GOMS model, incorporating the WM usage version of the generic command entry method. The intermediate versions produced in the separate passes above are discarded.

```
Method for goal: edit the document.
    Step 1. Get next unit task information from marked-up manuscript.
    Step 2. Decide: If no more unit tasks, then return with goal accomplished.
    Step 3. Accomplish goal: move to the unit task location.
    Step 4. Accomplish goal: perform the unit task.
    Step 5. Goto 1.

Selection rule set for goal: perform the unit task.
    If the task is moving text, then accomplish goal: move text.
    If the task is deletion, then accomplish goal: delete text.
    If the task is copying, then accomplish goal: copy text.
    ... etc. ...
    Return with goal accomplished.

Method for goal: move to the unit task location.
    Step 1. Get location of unit task from manuscript.
    Step 2. Decide: If unit task location on screen, return with goal
           accomplished.
    Step 3. Use scroll bar to advance text.
    Step 4. Goto 2.

Method for goal: move text.
    Step 1. Accomplish goal: cut text
    Step 2. Accomplish goal: paste text
    Step 3. Verify correct text moved.
    Step 4. Return with goal accomplished.

Method for goal: cut text.
    Step 1. Accomplish goal: select text.
    Step 2. Retain that the command is CUT, and accomplish goal: issue a
           command.
    Step 3. Return with goal accomplished.

Method for goal: paste text.
    Step 1. Accomplish goal: select insertion point.
    Step 2. Retain that the command is PASTE, and accomplish goal: issue a
           command.
    Step 3. Return with goal accomplished.

Selection rule set for goal: select text.
    If text-is word, then accomplish goal: select word.
    If text-is arbitrary, then accomplish goal: select arbitrary text.
    Return with goal accomplished.
Method for goal: select word.
    Step 1. Locate middle of word.
    Step 2. Move cursor to middle of word.
    Step 3. Double-click mouse button.
    Step 4. Verify that correct text is selected
    Step 5. Return with goal accomplished.
```

```
Method for goal: select arbitrary text.
    Step 1. Locate beginning of text.
    Step 2. Move cursor to beginning of text.
    Step 3. Press mouse button down.
    Step 4. Locate end of text.
    Step 5. Move cursor to end of text.
    Step 6. Verify that correct text is selected.
    Step 7. Release mouse button.
    Step 8. Return with goal accomplished.

Method for goal: select insertion point.
    Step 1. Locate insertion point.
    Step 2. Move cursor to insertion point.
    Step 3. Click mouse button.
    Step 4. Verify that insertion cursor is at correct place.
    Step 5. Return with goal accomplished.

Method for goal: issue a command.
    Assumes that user does not use command-key shortcuts
    Step 1. Recall command name and retrieve from LTM the menu name for it.
    Step 2. Recall menu name, and locate it on Menu Bar.
    Step 3. Move cursor to menu name location.
    Step 4. Press mouse button down.
    Step 5. Verify that menu appears.
    Step 6. Recall command name, and locate it in menu.
    Step 7. Move cursor to command name location.
    Step 8. Recall command name, and verify that it is selected.
    Step 9. Release mouse button.
    Step 10. Forget menu name, forget command name, and return with goal
           accomplished.
```

**Step C: Documenting and Checking the Analysis**

Table 1 shows the list of analyst-defined operators, and Table 2 shows the information contained in the task description for this GOMS model. The assumptions and judgment calls made are listed in Table 3.

As shown in Table 3 the methods assume that the user's hand is on the mouse and stays there throughout the editing tasks. If we identified which mouse moves had the property that the hand was on the keyboard, we could add the homing time to the time for that mouse move operator. We could do likewise for any keyboard keystroke times. A more elegant way would be to write a method for the move-cursor operator like this:

```
Method for goal: move the cursor to a specified location
    Step 1. Decide: If hand not on mouse, then move hand to mouse.
           (.4 sec homing time)
    Step 2. Move cursor to specified location.
           (typically 1.1 sec)
    Step 3. Return with goal accomplished.
```

Then the move-cursor operators in the example methods would be replaced with accomplishing the goal of moving the cursor, which would invoke this method.

**Table 1**
Analyst-Defined Operators for the Example

| |
|---|
| `Get next unit task information from marked-up manuscript` - look at manuscript and scan for the next edit marking, and put some of the task description into working memory. |
| `No more unit tasks` - tells whether there was another edit marking. |
| `Task is` ... - tells whether task is of the specified type, such as move, copy, etc. |
| `Get location of unit task from manuscript` - look at edit marking on manuscript and determine its position. |
| `If unit task location on screen` - tells whether the material corresponding to the edit marking is on the screen. |
| `Use scroll bar to advance text` - a high-level operator that could be expanded into a set of methods. |
| `Locate` - get information from task description, and map to perceptual location on screen. |
| `Text-is` - tells whether text is a word, sentence, or arbitrary. |
| `Verify` - compare results to goal to check that desired result is achieved. |
| `Move cursor to` - move mouse until cursor at specified point. |
| `Click mouse button.` |
| `Double-click mouse button.` |
| `Press mouse button.` |
| `Release mouse button.` |

**Table 2**
Example Task Description

| |
|---|
| `Task is to move specified piece of text` |
| `Piece of text is a word, or arbitrary` |
| `Position of beginning of text` |
| `Position of end of text, if it is arbitrary` |
| `Position of destination` |

**Table 3**
Example Assumptions and Judgment Calls

| |
|---|
| The topmost goal in the analysis is editing a document; it was assumed that there are no critical interactions with other aspects of the user's task environment. |
| The unit-task control structure is assumed for the topmost method. |
| Users view moving text on this system as first cutting, then pasting. |
| Users know that selecting text is done the same way everywhere in the system, and take advantage of it by having a subgoal and a method for it, and likewise for invoking commands. |
| The analysis has been simplified by ignoring the command key shortcuts for CUT and PASTE. |
| When selecting an arbitrary piece of text, users view as two separate steps deciding what the beginning point is and putting the cursor there. Likewise, deciding where the end point is and putting the cursor there are two separate steps. There are plausible alternative judgment calls. |
| The user's hand is on the mouse and stays there throughout the editing tasks. |

**Step D: Checking Sensitivity to Judgment Calls**

The following are some examples of how the sensitivity of the analysis to the judgment calls can be checked.

***Alternative view of the move task*.** Suppose the user does not decompose the move task into cut-then-paste as we did, but thinks of move as a single goal. Suppose you wrote out the methods according to this decomposition by providing a different move method that corresponds well to the user's decomposition. One possibility is that the user could select the text, issue a move command, and then click the mouse where the text is to be moved to. A Macintosh-like system could actually execute the command by cutting the text and then pasting it, but the user would issue only a move command. Clearly, if users like to think of a move this way, they probably would think of a copy this way as well.

With methods tailored to this alternative judgment of how move and copy goals decompose, what you might see is that more methods would be needed overall because we couldn't share the cut and paste submethods with other editing methods. This means that the editor might be harder to learn overall, due to more methods to learn. So the quality of the design in terms of its learnability and consistency is probably sensitive to whether our judgment call is correct. We may want to explore these alternatives by actually working out an alternative set of methods and comparing the two analyses or designs in detail.

If we stick with the original judgment call, it would be a good idea to be sure that the user decomposes the task the same way, into the cut-then-paste form. We could find out if users actually do this, or since it seems to be a reasonably natural way to view the task, we could try to encourage the user to look at it this way, perhaps by pointing out the advantages of this view in the documentation. This way we would have some confidence that the methods actually adopted by the user are like those we based the design on (see the Documentation discussion below). So our check of sensitivity suggests that the original judgment call was satisfactory.

***Effects of command shortcuts*.** As a second use of the analysis example, consider the simplifying assumption that the user would not use the command-key method of issuing commands on the Mac. Are our conclusions sensitive to this? Obviously both our learning time and execution time estimates will differ a lot depending on whether the user uses these shortcuts. If we are comparing two designs that differ in whether command-key shortcuts are available, this is actually not a question of a simplifying assumption, but of desirable features of the designs. But if we have simply assumed that the user will use the shortcuts in one of the designs, but not in the other, when they could be available in both, then the comparison of the designs will be seriously biased.

Consider the role of shortcuts in the above alternate analysis for move. We can conclude that the simplifying assumption that users would not use short-cuts would not influence the choice between the alternate designs for move. This is because the main differences between the designs will be how many commands have to be issued and how many methods have to be learned. Any differences will be reflected in the shortcuts the same as in the non-shortcuts, because the shortcuts have a one-to-one relationship with the full methods. So, for example, whichever design has the fewest methods to be learned, will also have the fewest shortcuts to be learned. So our check suggests that ignoring the shortcuts is not a bias on the particular choice of designs, but does produce much higher execution times.

## 5. USING A GOMS TASK ANALYSIS

Once the GOMS model analysis is completed (either for an existing system or one under design), it is time to make use of it to evaluate the quality of the design. Notice that if the model has been constructed during the design process itself, there is a good chance that some of the evaluation was in fact done on the fly; when a design choice ended up requiring a complex and difficult method, some change was probably made immediately.

The evaluation process described below will yield numbers and other indications of the quality of the design. As in any evaluation technique, these measures are easiest to make use of if there are at least two systems being compared. One of these systems might be an existing, competitive product. For example, IBM could have set the goal of making OS/2 for their PS line "as easy to learn and use as the Apple Macintosh." They could then have compared these measures for their new design with the measures from a GOMS model for the same tasks on a Macintosh to determine how close they were to this goal.

If heavy usage of a GOMS analysis is involved, consider implementing the methods as a computer program in order to automate checks for accuracy, and to generate counts of operators, steps, and so forth, used in the estimation procedures below.

### 5.1 Qualitative Evaluation of a Design

Several overall checks can be done that make use of qualitative properties of the GOMS model.

• Naturalness of the design - Are the goals and subgoals ones that would make sense to a new user of the system, or will the user have to learn a new way of thinking about the task in order to have the goals make sense?

• Completeness of the design - Construct the complete action/object table - are there any goals you missed? Check that there is a method for each goal and subgoal.

• Cleanliness of the design - If there is more than one method for accomplishing a goal, is there a clear and easily stated selection rule for choosing the appropriate method? If not, then some of these methods are probably unnecessary.

• Consistency of the design - By "consistency" is meant method consistency. Check to see that similar goals are accomplished by similar methods. Especially check to see that if there are similar subgoals, such as selection of various types of objects, then there are similar submethods, or better, a single submethod, for accomplishing the subgoals. The same idea applies for checking consistency between this and other systems - will methods that work on the other system also work on this one?

• Efficiency of the design - The most important and frequent goals should be accomplished by relatively short and fast-executing methods.

### 5.2 Predicting Human Performance

**What is Learned Versus What is Executed**

NGOMSL models can be used to predict the *learning time* that users will take to learn the procedures represented in the GOMS model, and the *execution time* users will take to execute specific task instances by following the procedures. It is critical to understand the difference between how these two usability measures are predicted from a GOMS model.

• *The total number and length of all methods determines the learning time.* The time to learn a set of methods is basically determined by the total length of the methods, which is given by the number of NGOMSL statements in the complete GOMS model for the interface. This is the amount of procedural knowledge that the user has to acquire in order to know how *to use the system for all of the possible tasks under consideration*.

• *The methods, steps, and operators required to perform a specific task determines the execution time.* The time required to accomplish a task instance is determined by the number and content of NGOMSL statements that have to be executed to get that specific task done. The time required by each statement is the sum of a small fixed time for the statement plus the time required by any external or mental operators executed in the statement.

There may be little relationship between the number of statements that have to be learned and the number of statements that have to be executed. The situation is exactly analogous to an ordinary computer program - the time to compile a program is basically determined by the program length, but the execution time in a particular situation can be unrelated to the length of the program; it depends on how many and which statements get executed.

Typically, performing a particular task involves only a subset of the methods in the GOMS model, meaning that the number of statements executed may be less than the number that must be learned. For example, you may know the methods for doing a lot of text editor commands, which might require a few hundred NGOMSL statements to describe. But to perform the task of deleting a character will only involve executing a few of those statements. On the other hand, performing a task might involve using a method repeatedly, or looping on a set of steps (e.g., the top-level unit task method, which loops, doing one unit task after another). In this case, the number of statements executed in order to perform a task might easily be much larger than the number of statements in the GOMS model.

So, in estimating learning time, count how many NGOMSL statements that the user has to learn in order to know how to use the system for all possible tasks of interest. In estimating execution time for a specific task, determine which and how many statements have to be passed through in order to perform the task.

**Estimate Times at the Standard Primitive Operator Level of Detail**

A useful feature of GOMS models is the they can represent an interface at different levels of detail. However, a GOMS model can predict learning and execution time sensibly only if the lowest-level operators used in the model are ones (1) that you can reasonably assume that the user already knows how to do, and (2) for which stable time estimates are available. A keystroke is a good example of an operator that a user is typically assumed to already know, and which is executed in a predictable amount of time. If a method involves only such standard primitive operators (see section 2.2), and the user already knows them, the time to learn a method depends just on how long it takes to learn the content and sequence of the method steps. Likewise, the time to execute the method can be predicted by adding up a standard execution time for each NGOMSL statement in the method plus the predicted execution time for each standard primitive operator executed in the method steps, plus the execution time assigned to any analyst-defined operators.

In contrast, if you have a GOMS model which is written just at the level of high-level operators that the user has to learn how to perform, then the learning time estimates have to be poor because the learning times for the operators will be relatively large and probably unknown. Typically, the execution time for operators at this level will also be fairly large and unknown. For example, consider the following GOMS model for the task of writing computer programs:

```
Method for goal: write computer program
    Step 1. Invent algorithm.
    Step 2. Code program.
    Step 3. Enter code into computer.
    Step 4. Debug program.
    Step 5. Return with goal accomplished.
```

Suppose our model stops with these very high-level operators, and we don't believe that users already know how to debug a program or invent algorithms. Estimating the learning time as a simple function of the length of this

method is obviously absurd; it will take grossly different times to learn how to execute the operators in each of these steps, and these times will be very long. Likewise, these operators have grossly different, unknown, and relatively long execution times. In contrast, the typical primitive external operators such as pressing a key, or relatively simple mental operators such as looking at a manuscript, can be assumed to be already known to the user, and have relatively small, constant, and known execution times. Thus, if the methods have been represented at the standard primitive operator level, the calculations presented below for predicting learning and execution times will produce useful results.

**Estimating Learning Time**

**Pure vs. total learning time.** In estimating learning time, we might be interested in the total time needed to complete some training process in which users learn the methods in the context of performing some tasks with the system, perhaps with some accompanying reading or other study. This total time consists of the time to *execute the training tasks* in addition to the time required to learn how to perform the methods themselves, which is the *pure learning time*. This pure learning time has two components, the *pure method learning time*, and the *LTM item learning time*, which is the time required to memorize items that will be retrieved from LTM during method execution. Once the methods and LTM items are learned, the same training situation could be executed much more quickly. Thus the pure learning time represents the excess time required to perform the training situation due to the need to learn the methods. This pure learning time can be estimated as shown below. If the procedures to be used in training are known, it may be useful to estimate the total learning time by adding the time required to execute the training procedures. Thus, in summary:

*Total Learning Time = Pure Method Learning Time + LTM Item Learning Time + Training Procedure Execution Time.*

***Pure method learning time.*** Kieras & Bovair (1986) and Bovair, Kieras, & Polson (1990) found that pure learning time was proportional to the number of production rules that had to be learned. NGOMSL was defined so that one NGOMSL statement corresponds to one production rule, and so the pure learning time can be estimated from NGOMSL statements as well. But the time required to learn a single NGOMSL statement depends of specifics of the learning situation, such as what the criterion for learning consists of. The Bovair, Kieras, & Polson work used a very rigorous training situation, while Gong (1993; see also Gong & Kieras, 1994) measured learning time in a much more realistic training situation.

The Bovair, Kieras, & Polson *rigorous* learning situation was as follows: The methods correspond to new or novice users, not experts. The methods are explicitly presented; the learner does not engage in problem-solving to discover them. Efficient presentation and feedback, as in computer-assisted instruction, are used, rather than "real-world" learning approaches. The methods are presented one at a time, and are practiced to a certain criterion, such as making a set of deletions perfectly, before proceeding to next method. The total training time was defined to be the total time required to complete the training on each method.

The Gong *typical* training situation was much more realistic: The users went through a demonstration task accompanied by a verbal explanation, and then performed a series of training task examples. The total training time was defined as the time sum of the time required for the demonstration and the training examples.

Thus, to estimate the pure method learning time, decide whether the rigorous or typical situation is relevant, and calculate:

*Pure Method Learning Time = Learning Time Parameter $\times$ Number of NGOMSL statements to be learned*

Where:

*Learning Time Parameter = 30 sec for rigorous procedure training*
*17 sec for a typical learning situation*

Note that if the user already knows some of the methods, either from previous training or experience, or from having learned a different system which uses the same methods, then these methods should not be included in the count of NGOMSL statements to be learned.

***LTM item learning time.*** If many Retrieve-from-LTM operators are involved, then we would predict that learning will be slow due to the need to memorize the information that has to be retrieved when executing the methods. We can estimate how long it will take to store the information in LTM, using the Model Human Processor parameters (Card, Moran, & Newell, 1983, Ch.2), which is a value of about 10 sec/chunk for LTM storage time. Gong (1990) obtained results in a realistic training situation that suggest a value of 6 sec/chunk, which is the recommended value.

There is no established and verified technique for counting how many chunks are involved in to-be-memorized information, so the suggestions here should be treated as heuristic suggestions only. Count the number of chunks in an item with judgment calls as follows:

- one chunk for each familiar pattern in the retrieval cue
- one chunk for each familiar pattern in the retrieved information
- one chunk for the association between the retrieval cue and the retrieved information

For example, suppose the to-be-stored association for a command is *move cursor right by a word is ctrl-right arrow*. Then:

(move cursor right) (by a word) = 2 chunks for retrieval cue
(ctrl) (right-arrow) = 2 chunks for retrieved information
association between the two = 1 chunk

For a total of 5 chunks, or 50 sec minimum pure learning time. Add this estimated time to the total learning time.

Do not count LTM storage time if the item is already known. For example, the above example is a common convention on PCs, and so would probably be known to an experienced PC user.

***Estimating gains from consistency.*** If the design is highly consistent in the methods, the new user will be able to learn how to use it more easily than if the methods are not consistent with each other. One sign of a highly consistent interface is that there are generic methods (see the Section 1.2 example, and the issue-command method in the above example) that are used everywhere they are appropriate, and few or no special case methods are required. So, if a GOMS model for a user interface can be described with a small number of generic methods, it means that the user interface is highly consistent in terms of the method knowledge. Thus, describing generic methods, where they exist, is a way to "automatically" take this form of interface consistency into account.

However, sometimes the methods can be similar, with only some small differences; the research suggests that this cruder form of consistency reduces learning time as well, due to the *transfer of training* from one method to another. Kieras, Bovair, & Polson suggested a theoretical model for the classic concept of common elements transfer of training (Kieras & Bovair, 1986; Polson, 1987; Bovair, Kieras, & Polson, 1990). These transfer gains can be estimated by identifying similar methods and similar NGOMSL statements in these methods, and then deducting the number of these similar statements from the above estimate.

The criterion for "similar" can be defined in a variety of ways. Here will be described a relatively simple definition based on the Kieras & Bovair (1986) model of transfer. In summary, consistency can be measured in terms of how many statements have to be modified to turn one method into another, related, method. Only a very simple modification is allowed. If two statements can be made identical with this modification, then the statements are classified as "similar." After learning the first of two similar statements, the user can learn the second one so easily that to a good first approximation, there is no learning time required for the second statement at all. The same procedure applies to both methods and selection rules.

More specifically, the procedure for estimating transfer is as follows:

**1. Find candidates for transfer.** Find the methods that might be similar to each other. You do this by finding two methods that have similar goals. Let's call them method A and method B. Normally the two methods will not be completely identical because the goals they accomplish will be different. If both the verb and the noun in the `<verb noun>` goal descriptions are different, the methods are not similar at all, and there will be no consistency gains between them. If they are different on only one of the terms (e.g. `MOVE TEXT` versus `COPY TEXT`) the methods are similar enough to proceed to determining how many statements are similar in the two methods.

**2. Generalize method goals.** To determine which statements in method A and method B are similar, generalize the goals of the two methods by identifying the single term in the goal specifications of the two methods that is different and change it to a "parameter." Make the corresponding change throughout both methods. What you have constructed is the closest generalization between the two methods. If they are identical at this point, it means you could have in fact defined a generic method for the two different goals; do so and go on look for other similar methods.

**3. Count similar statements.** If the two methods are not identical, start at the beginning of these two methods and go down through the method statements and the steps and count the number of NGOMSL statements that are identical in the two methods; stop counting when you encounter two statements that are non-identical. That is, once the flow of control diverges, there is no longer any additional transfer possible (according to the theoretical model). If the `Method` statement is the only one counted as identical, then there are no real similarities; the number of similar statements is zero, and there are no consistency savings between the two methods. But if there are some identical statements in addition to the `Method` statement, include the `Method` statement in the count.

**4. Deduct similar statements from learning time.** The identical statements counted this way are the ones classified as "similar." According to the model of transfer of training, only the first appearance of one of these similar statements requires full learning; the ones encountered later come "free of charge." Subtract the number of the similar statements from the total number of statements to be learned.

It is important to ensure that steps deemed identical actually contain or imply identical operators. The best way to ensure this is for the GOMS model to be worked out down to the level of primitive external operators such as keystrokes; the rigor at this level ensures that the higher-level methods will appear to be similar only if they invoke submethods that are in fact similar.

As an example of a transfer calculation, refer to the Final GOMS Model of the above example. Note that the two selection methods have some overlap and similarity:

```
Method for goal: select word.
    Step 1. Locate beginning of word.
    Step 2. Move cursor to beginning of word.
    Step 3. Double-click mouse button.
    Step 4. Verify that correct text is selected
    Step 5. Return with goal accomplished.

Method for goal: select arbitrary text.
    Step 1. Locate beginning of text.
    Step 2. Move cursor to beginning of text.
    Step 3. Press mouse button down.
    Step 4. Locate end of text.
    Step 5. Move cursor to end of text.
    Step 6. Verify that correct text is selected.
    Step 7. Release mouse button.
    Step 8. Return with goal accomplished.
```

The object in the goal specifications in the `Method` statements can be replaced with a parameter, `GOAL-OBJECT`, and the corresponding changes made throughout; these are our most generalized version of the two methods.

```
Method for goal: select GOAL-OBJECT.*
    Step 1. Locate beginning of GOAL-OBJECT.*
    Step 2. Move cursor to beginning of GOAL-OBJECT.*
    Step 3. Double-click mouse button.
    Step 4. Verify that correct text is selected
    Step 5. Return with goal accomplished.

Method for goal: select GOAL-OBJECT.*
    Step 1. Locate beginning of GOAL-OBJECT.*
    Step 2. Move cursor to beginning of GOAL-OBJECT.*
    Step 3. Press mouse button down.
    Step 4. Locate end of GOAL-OBJECT.
    Step 5. Move cursor to end of GOAL-OBJECT.
    Step 6. Verify that correct text is selected.
    Step 7. Release mouse button.
    Step 8. Return with goal accomplished.
```

We start counting identical statements from the beginning of the two methods. The `Method` statements, Step 1, and Step 2 are identical, but Step 3 is different, so we stop counting, giving a total of 3 similar statements (the `Method` statements, Step 1, and Step 2) for the two methods. The statements counted as similar are marked with an asterisk above.

The gain from consistency can now be estimated. Suppose the select-word method is learned first, requiring learning 6 NGOMSL statements, followed by learning the select-arbitrary-text method, which has a total of 9 statements. According to the transfer model, the user will have to work at learning the similar statements only the first time, while learning how to select words, and so will have to learn only Steps 3 through 8 of the select-arbitrary-text method. Thus, instead of a total learning time for the two methods of $(6 + 9) \times 30$ sec = 450 sec above the baseline, the estimated learning time will be only $(6 + 6) \times 30$ sec = 360 sec above the baseline, a substantial gain due to consistency.

As a negative example, consider the apparent similarity between the method for cutting text and the method for pasting text. Both seem to involve first selecting something, and then issuing a command based on the goal:

```
Method for goal: cut text
    Step 1. Accomplish goal: select text.
    Step 2. Accomplish goal: issue CUT command.
    Step 3. Return with goal accomplished.

Method for goal: paste text
    Step 1. Accomplish goal: select insertion point.
    Step 2. Accomplish goal: issue PASTE command.
    Step 3. Return with goal accomplished.
```

The verb in the goal specifications in the method statements can be generalized with a parameter, `GOAL-VERB`, and the corresponding changes made:

```
Method for goal: GOAL-VERB text
    Step 1. Accomplish goal: select text.
    Step 2. Accomplish goal: issue GOAL-VERB command.
    Step 3. Return with goal accomplished.

Method for goal: GOAL-VERB text
    Step 1. Accomplish goal: select insertion point.
    Step 2. Accomplish goal: issue GOAL-VERB command.
    Step 3. Return with goal accomplished.
```

But, counting from the beginning, we see that the two methods diverge immediately at Step 1; selecting text is not identical with selecting the insertion point; generalizing the goal doesn't deal with this fundamental difference. So, according to the transfer model, there are no similar statements; the `Method` statements by themselves do not count. Accordingly, there is no transfer of learning between these two methods, and the user will have to pay the full cost of learning two separate methods.

**Estimating Execution Time**

***Time estimate calculation.*** Estimating execution time is very similar to the Keystroke-Level Model approach (Card, Moran & Newell, 1983, Ch. 8). The time to execute a method depends on the time to execute the operators and on the number of cognitive steps, or production rules, involved. NGOMSL has been defined so that the NGOMSL statements defined above each correspond to a production rule that is executed. The execution time can only be estimated for specific task instances because only then will the number and sequence of steps and operators be determined.

To estimate the execution time for a task, choose one or more representative task instances. Execute the method (e.g. by hand) to accomplish each task instance, and record a trace of the NGOMSL statements and operators executed. Examine the trace and tabulate the statistics referred to below. The estimated execution time is given by:

*Execution Time = NGOMSL statement time + Primitive External Operator Time + Analyst-defined Mental Operator Time + Waiting Time*

*NGOMSL Statement Time = Number of statements executed $\times$ 0.1 sec*

*Primitive External Operator Time = Total of times for primitive external operators*

*Analyst-Defined Mental Operator Time = Total of times for mental operators defined by the analyst*

*Waiting Time = Total time when user is idle while waiting for the system*

Thus, each NGOMSL statement takes 0.1 sec to execute, whatever operator times are involved is additional. Note that the time to execute the built-in primitive mental operators is bundled into the NGOMSL statement execution time. The time of 0.1 sec is based on the assumption that each NGOMSL statement is actually implemented as a single production rule, and on the results of modeling work which assumes that the cognitive processor has a production rule architecture. When such models are fitted to task execution time data, the resulting estimate of cycle time is 0.1 sec (see Bovair, Kieras, & Polson, 1990; Kieras, 1986). This value is consistent with, though somewhat slower than, the Card, Moran, and Newell (1983, Ch. 2) figure for cognitive processor cycle time.

The *primitive external operator time* can be estimated using the Keystroke-Level Model values from Card, Moran, and Newell (1983, Ch. 8). These times can be refined by using empirical results or more exact calculations, such as applying Fitts' Law to hand or mouse movement times.

Gong (1990; see also Gong & Kieras, 1994) found that many of the mouse moves in a Macintosh interface were much more accurately estimated with Fitts' Law than with the Keystroke Level Model time of 1.1 sec. For example, the time to move to select a window was only about 0.2 sec (a very large target) and moves within a dialog box were also very fast, e.g. 0.1 sec, (a short distance). The Keystroke Level Model estimate of 1.1 sec is actually based on large moves to small targets, as in document text editing. Instead, Gong used values calculated from Fitts' Law (see Card, Moran, & Newell, 1983; p. 242), and considerably increased the accuracy of the execution time prediction.

The *analyst-defined* mental operator time, in the lack of any other information, can be estimated as 1.2 sec for each analyst-defined mental operator (see Olson & Olson, 1989). Some of the operators normally represented as taking a mental operator time should be treated differently, depending on the whether the methods they appear in are assumed to be well-known to the user. With experienced Macintosh users learning a new Macintosh application, Gong (1990) found that certain operators in the standard Macintosh methods should be assigned zero execution time, apparently because an experienced use can overlap these operators with other activities. Thus, if an experienced user is assumed, estimate zero time for the following operators in well-practiced methods:

- a `Locate` that is followed by a `Point`
- a `Verify`
- a `Wait`

The *waiting time* is the time when the user is waiting, idle, for the system's response. This is typically negligible in echoing keyboard input, but can be substantial for other operations. You can omit this if you are concerned only with the user interface, and are willing to say that either the system will be fast enough, or it is somebody else's problem if it isn't! For an existing system, you can measure these times directly by executing the task instance on the system. For a system under design, you could estimate these times by measuring comparable operations on similar systems.

**Mental Workload**

Less is known about the relationship between GOMS models and mental workload than for the learning and execution times, so these suggestions are rather speculative. One aspect of mental workload is the user's having to keep track of where he or she is in the "mental program" of the method hierarchy. Using a specific task instance, one can count the depth of the goal stack as the task is executed. The interpretation is not clear, but greater peak and average depth is probably worse than less. Another aspect of mental workload is Working Memory load; quantifying this requires making WM usage in the methods explicit as described above. One could count the peak number of chunks that are `Retain`ed before a `Forget`; this is a measure of how much has to be remembered at the same time. It seems reasonable to expect trouble if more than 5 chunks have to be maintained in WM at once. Lerch, Mantei, and Olson (1989) reported results suggesting that errors are more likely to happen at peak memory loads as determined by a GOMS analysis.

A final aspect of mental workload is less quantifiable, but is probably of considerable importance in system design. This is whether the user has to perform complex mental processing that is not part of interacting with the system itself, but is required in order to do the task using the system. These processes would typically be those that were bypassed in the analysis. If there are many complex analyst-defined mental operators, and they seem difficult to execute, one could predict that the design will be difficult and error-prone to use. The evaluation problem comes if the two systems being compared differ substantially in the bypassing operators involved.

For example, consider the task of entering a table of numbers in a document preparation system using tabs to position the columns (cf. Bennett, Lorch, Kieras and Polson, 1987). A certain document preparation system is not a WYSIWYG ("what you see is what you get") system, and so to get tab settings in the correct positions, the user has to tediously figure out the actual column numbers for each tab position, and include them in a tab setting command. The analyst could represent this complex mental process with a bypassing operator like `Figure-out-tab-setting-column`. After executing this operator, the user simply types the resulting column number in the tab setting command. However, on a simple WYSIWYG system the user can arrive at the tab positions easily by trial and error, using operators like `Choose-new-setting` and `Does-setting-look-right`, which apparently would be considerably simpler to execute than `Figure-out-tab-setting-column`. But the trial and error method on the WYSIWYG involves more keystrokes, mouse moves, and so on, than simply typing in the tab column numbers. Which system will have the longer execution time? Clearly, this cannot be answered without getting estimates of the true difficulty and time required for these operators.

### 5.3 Suggestions for Revising the Design

After calculating performance estimates, the analyst can revise the design, and then recalculate the estimates to see if progress has been made. Some suggestions for revising the design are as follows (cf. Card, Moran, & Newell, 1983, Ch 12):

- Ensure that the most important and frequent goals can be accomplished by relatively easy to learn and fast-executing methods. Redesign if rarely accomplished tasks are simpler than frequent ones.

- Try to reduce learning time by eliminating, rewriting, or combining methods, especially to get consistency. Examine the action/object table for ideas - goals involving the same action should have similar methods.

- If a selection rule can not be stated clearly and easily, then consider eliminating one or more of the alternative methods. If there is not a clear occasion for using it, it is probably redundant.

- Eliminate the need for `Retrieve-from-LTM` operators, which indicate that the user has to memorize information, and which will result in slow performance until heavily practiced.

- If there are WM load problems, see if the design can be changed so the user needs to remember less, especially if the system can do the remembering, or if key information can be kept on the display until the user needs it.

- Modify the design to eliminate the need for the user to execute high-level complex mental operators, especially if they involve a slow and difficult cognitive process.

- The basic way to speed up execution time is to eliminate operators by shortening the methods. But notice that complex mental operators are usually much more time-consuming than simple

motor actions, and so it can be more important to reduce the need for thinking than to save a few keystrokes. So do not reduce the number of keystrokes if an increase in mental operators is the result.

### 5.4 Using the Analysis in Documentation

The GOMS model is supposed to be a complete description of the procedural knowledge that the user has to know in order to perform tasks using the system. If the methods have been tested for completeness and accuracy, the procedural documentation can be checked against the methods in the GOMS model. Any omissions and inaccuracies should stand out. Alternatively, the first draft of the procedure documentation could be written from the GOMS model directly, as a way to ensure completeness and accuracy from the beginning. A similar argument can be made for the use of a GOMS model in specifying the content of On-line Help systems (see Elkerton, 1988).

Notice that if the documentation does not directly provide the required methods and selection rules to the user, the user is forced to deduce them. Sometimes this is reasonable; it should not be necessary to spell out every possible pathway through a menu system, for example. But it is often the case that even "good" training documentation presents methods that are seriously incomplete and even incorrect (cf. Kieras, 1990), and selection rules are rarely described.

While the GOMS analysis clearly should be useful in specifying the content of documentation, it offers little guidance concerning the form of the documentation, that is, the organization and presentation of procedures in the document. One suggestion that does stand out (cf. Elkerton, 1988) is to ensure that the index, table of contents, and headings are organized by user's goals, rather than function name, to allow the user to locate methods given that they often know their natural goals, but not the operators involved. Elkerton and his co-workers (Elkerton & Palmiter, 1991; Gong & Elkerton, 1990) provide more details and experimental demonstrations that GOMS-based documentation and help is markedly superior to conventional documentation and help.

## ACKNOWLEDGEMENTS

Many helpful comments on earlier forms of this material have been contributed by Jay Elkerton, John Bennett, and several students. Thanks are especially due to the students in my Fall 1986 seminar on "The Theory of Documentation" who got me started on providing a procedure and clear notation for a GOMS analysis. Comments on this version of the procedure will be very much appreciated. The research that underlies this work was originally supported by IBM and the Office of Naval Research.

## REFERENCES

Anderson, J.R. (1982). Acquisition of cognitive skill. *Psychological Review*, **89**, 369-406.

Bennett, J.L., Lorch, D.J., Kieras, D.E., & Polson, P.G. (1987). Developing a user interface technology for use in industry. In Bullinger, H.J., & Shackel, B. (Eds.), Proceedings of the Second IFIP Conference on Human-Computer Interaction, *Human-Computer Interaction - INTERACT '87*. (Stuttgart, Federal Republic of Germany, Sept. 1-4). Elsevier Science Publishers B.V., North-Holland, 21-26.

Bjork, R.A. (1972). Theoretical implications of directed forgetting. In A.W. Melton and E. Martin (Eds.), *Coding Processes in Human Memory*. Washington, D.C.: Winston, 217-236.

Bovair, S., Kieras, D.E., & Polson, P.G. (1990). The acquisition and performance of text editing skill: A cognitive complexity analysis. *Human-Computer Interaction,* **5**, 1-48.

Butler, K. A., Bennett, J., Polson, P., and Karat, J. (1989). Report on the workshop on analytical models: Predicting the complexity of human-computer interaction. *SIGCHI Bulletin*, 20(4), pp. 63-79.

Byrne, M.D., Wood, S.D, Sukaviriya, P., Foley, J.D, and Kieras, D.E. (1994). Automating Interface Evaluation. In *Proceedings of CHI*, 1994, Boston, MA, USA, April 24-28, 1994). New York: ACM, pp. 232-237.

Card, S.K., Moran, T.P., & Newell, A. (1980a). The keystroke-level model for user performance time with interactive systems. *Communications of the ACM* , 23(7), 396-410.

Card, S., Moran, T. & Newell, A. (1983). *The Psychology of Human-Computer Interaction*. Hillsdale, New Jersey: Erlbaum.

Diaper, D. (Ed.) (1989). *Task analysis for human-computer interaction*. Chicester, U.K.: Ellis Horwood.

Elkerton, J. (1988). Online Aiding for Human-Computer Interfaces. In M. Helander (Ed.), *Handbook of Human-Computer Interaction* (pp. 345-362). Amsterdam: North-Holland Elsevier.

Elkerton, J., & Palmiter, S. (1991). Designing help using the GOMS model: An information retrieval evaluation. *Human Factors*, **33**, 185-204.

Gong, R. J. (1993). *Validating and refining the GOMS model methodology for software user interface design and evaluation.* PhD Dissertation, University of Michigan.

Gong, R. & Elkerton, J. (1990). Designing minimal documentation using a GOMS model: A usability evaluation of an engineering approach. In *Proceedings of CHI'90, Human Factors in Computer Systems* (pp. 99-106). New York: ACM.

Gong, R., & Kieras, D. (1994). A Validation of the GOMS Model Methodology in the Development of a Specialized, Commercial Software Application. In *Proceedings of CHI*, 1994, Boston, MA, USA, April 24-28, 1994). New York: ACM, pp. 351-357.

Gould, J. D. (1988). How to design usable systems. In M. Helander (Ed.), *Handbook of human-computer interaction*. Amsterdam: North-Holland. 757-789.

Gray, W. D., John, B. E., & Atwood, M. E. (1993). Project Ernestine: A validation of GOMS for prediction and explanation of real-world task performance. *Human-Computer Interaction*, **8**, 3, pp. 237-209.

John, B. E. & Kieras, D. E. (1994) The GOMS family of analysis techniques: Tools for design and evaluation. Carnegie Mellon University School of Computer Science Technical Report No. CMU-CS-94-181. Also appears as the Human-Computer Interaction Institute Technical Report No. CMU-HCII-94-106.

Kieras, D.E. (1986). A mental model in user-device interaction: A production system analysis of a problem-solving task. Unpublished manuscript, University of Michigan.

Kieras, D.E. (1988). Making cognitive complexity practical. In *CHI'88 Workshop on Analytical Models*, Washington, May 15, 1988.

Kieras, D.E. (1988). Towards a practical GOMS model methodology for user interface design. In M. Helander (Ed.), *Handbook of Human-Computer Interaction* (pp. 135-158). Amsterdam: North-Holland Elsevier.

Kieras, D.E. (1990). The role of cognitive simulation models in the development of advanced training and testing systems. In N. Frederiksen, R. Glaser, A. Lesgold, & M. Shafto (Eds.), *Diagnostic Monitoring of Skill and Knowledge Acquisition*. Hillsdale, N.J.: Erlbaum.

Kieras, D. (1994). GOMS Modeling of User Interfaces using NGOMSL. Tutorial Notes, CHI'94 Conference on Human Factors in Computer Systems, Boston, MA, April 24-28, 1994.

Kieras, D. (in press). Task Analysis and the Design of Functionality. To appear in *Handbook of Computer Science and Engineering*. Boca Raton, Florida: CRC Press.

Kieras, D. E., & Bovair, S. (1984). The role of a mental model in learning to operate a device. *Cognitive Science,* 8, 255-273.

Kieras, D.E., & Bovair, S. (1986). The acquisition of procedures from text: A production-system analysis of transfer of training. *Journal of Memory and Language*, **25**, 507-524.

Kieras, D.E. & Polson, P.G. (1985). An approach to the formal analysis of user complexity. *International Journal of Man-Machine Studies*, **22**, 365-394.

Kieras, D.E., Wood, S.D., Abotel, K., & Hornof, A. (1995). GLEAN: A Computer-Based Tool for Rapid GOMS Model Usability Evaluation of User Interface Designs. In Proceeding of UIST, 1995, Pittsburgh, PA, USA, November 14-17, 1995. New York: ACM. pp. 91-100.

Kirwan, B., & Ainsworth, L. K. (1992). *A guide to task analysis.* London: Taylor and Francis.

Landauer, T. (1995). *The trouble with computers: Usefulness, usability, and productivity.* Cambridge, MA: MIT Press.

Lerch, F.J., Mantei, M.M., & Olson, J. R., (1989). Skilled financial planning: The cost of translating ideas into action. In *CHI'89 Conference Proceedings*, 121-126.

Lewis, C. & Rieman, J. (1994) *Task-centered user interface design: A practical introduction.* Shareware book available at ftp.cs.colorado.edu/pub/cs/distribs/clewis/HCI-Design-Book

Nielsen, J. & Mack, R.L. (Eds). (1994). *Usability inspection methods*. New York: Wiley.

Olson, J. R., & Olson, G. M. (1989). The growth of cognitive modeling in human-computer interaction since GOMS. Technical Report No. 26, Cognitive Science and Machine Intelligence Laboratory, University of Michigan, November, 1989.

Polson, P.G. (1987). A quantitative model of human-computer interaction. In J.M. Carroll (Ed.), *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction.* Cambridge, MA: Bradford, MIT Press.