A Guide to GOMS Model Usability Evaluation using GOMSL and GLEAN4

Revision: March 31, 2006

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: (734) 763-6739; Fax: (734) 763-1260
Email: kieras@eecs.umich.edu
Anonymous ftp for documents: ftp.eecs.umich.edu people/kieras

This document is a heavily modified version of the earlier "Guides" to GOMS modeling Kieras (1988, 1997a), and supersedes the 1999 Guide referring to GLEAN3

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 **G**oals, **O**perators, **M**ethods, and **S**election 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 Kieras(1988, 1997a), 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. 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. 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.

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.

This document presents an executable form of NGOMSL, called **GOMSL** (**GOMS L**anguage) which is processed and executed by a GOMS model simulation tool, **GLEAN4** (**GOMS L**anguage **E**valuation and **An**alysis). GLEAN4 is an extended version of GLEAN3, and was inspired by the original GLEAN tool developed by Scott Wood (1993; see also Byrne, Wood, Sukaviriya, Foley, & Kieras, 1994) and reimplemented and elaborated in Kieras, Wood, Abotel, & Hornof (1995). GLEAN3 differs in three main ways from the earlier versions: (1) GLEAN3 is implemented in C++ rather than the original LISP, for better performance and interoperability; (2) GLEAN3's

version of GOMSL is more powerful and better-defined; (3) Unlike the earlier versions, GLEAN3 is based on a comprehensive cognitive architecture, namely a simplified version of the EPIC architecture for simulating human cognition and performance (Kieras & Meyer, 1997). GLEAN3 uses GOMSL for representing the procedural knowledge rather than the technically more difficult production-rule representation native to EPIC. GLEAN4 differs from GLEAN3 in the following ways: (1) GLEAN4 incorporates some additional memory operators suitable for representing working memory for task information; (2) GLEAN4 supports multithreaded methods, including a constantly-running thread for detecting and responding to interrupts; (3) GLEAN4 provides an initial set of mechanisms for representing error detection and handling, based on Wood's (2000) analysis of adding error-handling to GOMS models. (4) GLEAN4 implements some basic operators for constructing executable simulations of High-Level GOMS models for analysis of functionality design prior to interface design (see Kieras, 2004). For simplicity, in the rest of this document, the current version of the tool will be referred to as simply GLEAN, with no version qualification.

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, 1996a, b) 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 (1997).

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, 1996a). 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 (1996a) 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; GLEAN was developed to remove this obstacle, but of course, additional effort is required to express the GOMS model precisely enough for a computer-based tool to use it.

What is a GOMS 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 (1997) 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 are incredibly complex; 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 The Cognitive Architecture for GOMSL

GOMSL runs in a simple cognitive architecture, implemented in the GLEAN simulation environment, that represents the assumed mechanisms and capabilities of the human information processing system. One way to view this architecture is that it is a computationally realized version of the Model Human Processor (Card, Moran, & Newell, 1983), but in fact it is rather more sophisticated, being based more on the EPIC architecture (Kieras & Meyer, 1997). GLEAN at this time is considered to be incomplete and subject to extension and elaboration in the future to more fully incorporate the mechanisms developed for EPIC.

Figure 1 shows the basic architectural structure assumed in GOMSL and its current GLEAN implementation. There is a simulated human made up of several processors, and a simulated device - the system that the simulated human is interacting with. There are perceptual processors (visual and auditory) and motor processors (manual and vocal). The cognitive processor executes methods written in GOMSL in order to accomplish goals, using information from the perceptual processors, Long-Term Memory (LTM), and the specific properties of the tasks under execution, called the Task Instance Descriptions.

The simulated device appears in Figure 1 under the label of "Device Behavior Simulation" to make it clear that it is only the visible behavior of the device that is relevant, and only to the extent that the GOMS methods specify the behavior of the device. Thus, the device need not be an actual system or application, but only the behavioral mimic of one at the level of detail represented in the GOMS model. In fact, the device can be a dummy device that does nothing, in which case the GOMS methods cannot depend on how the device responds to user input. But a device of

arbitrary complexity can be connected to the simulated human by replacing the dummy device with a module that responds to input from the simulated human user by supplying simulated output to the user, at whatever detail or fidelity is desired. In addition, multi-human multi-device simulations can be constructed, for example, by having two simulated humans interacting with each other through the auditory/vocal channels, and each human interacting with a device through the visual/manual channels.

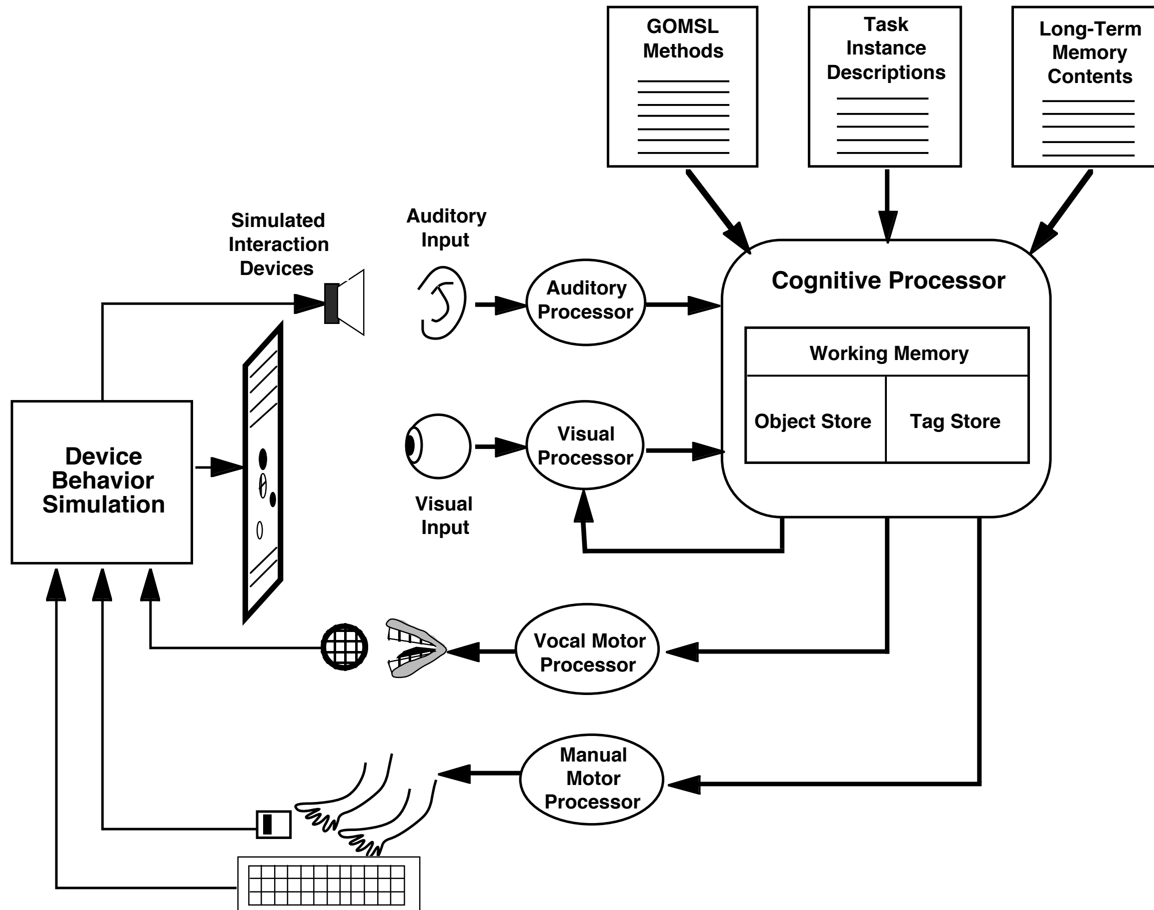

Figure 1. The GLEAN4 architecture.

In overview, various data in the system are represented in the form of objects which have properties and a value for each property. The various processors modify object properties in ways that will be described below in the context of the different operators used in GOMSL methods to interact with the processors. The auditory processor accepts either speech or sound inputs and makes them available to the cognitive processor, where they are tested or waited for by auditory operators. The visual processor accepts visual events that take the form of an object appearing with specified properties, or a change in a visual property of an existing visual object. Visual input is either searched or waited for with visual operators. Vocal output is produced when a vocal operator is executed, specifying an utterance to be produced, which is sent to the device or another simulated human auditory processor. Manual output takes many different forms depending on the type of movement specified by a manual motor operator.

The architecture contains certain memory structures. Each perceptual processor has its own working memory. The cognitive processor's working memory has two partitions: an object store and a tag store. The tag store represents the conventional human working memory, and contains tags (labels) each with an associated value. GOMSL

operators can add or remove tag-value pairs from working memory. Wherever a tag is used in a GOMSL expression, it will be replaced with the associated value during execution.

The object store contains a set of objects that are currently "in focus" in working memory, meaning that their properties are immediately available to GOMS methods. For example, when the visual processor is "looking" at a particular visual object, then that object is "in focus" in visual working memory, and so all of its properties, such as its color, are immediately available. Likewise, as long as an speech object is present in auditory working memory, it is in focus. The visual processor is limited to having only one object in focus at a time, corresponding to a simple representation of the visual system as having the eye fixated on only one object at a time. If the eye is moved to another object, the properties of the previously in-focus object become unavailable. The auditory processor can hold any number of objects in focus, but in practice, auditory objects decay from auditory working memory too rapidly for there to be more than a few present at a time.

The Task Instance Description and Long-Term Memory are similar to the perceptual processors, and in fact each is represented by a processor with a memory that contains objects with property values, which are accessed and examined by operators in GOMSL methods. However, unlike the perceptual processors, the contents of Long-Term Memory is static, being specified at the time the GOMS model is compiled, and likewise the contents of the Task Instance Description are static, unless special operators are used to modify it. Like the visual processor , the task processor and long-term memory processors are limited to one in-focus object at a time, corresponding to the information about only one task or long-term memory item being active at a time in working memory.

Finally, the cognitive processor executes the GOMS methods using the same kind of approach as a conventional computer programming language. The command to accomplish a goal causes the current method state to be pushed onto a run-time stack, and then the method for the new goal begins execution with the first step, and continuing with the next step in the method. When a method returns, the stack is popped and the previous method state is restored. It is important to note, however, that the memory is assumed to be global - there are no "local variables" allocated, saved, and restored on the stack, as in conventional programming languages. GLEAN4 allows multiple execution threads to be running simultaneously, but within each thread, the method hierarchy executes in the same way, and all memory contents are shared between the threads.

1.3 An Example GOMS Analysis

Before presenting the details of the GLEAN architecture, GOMSL syntax, and the GOMS modeling method, it is useful to examine a sample GOMS analysis. The tasks and systems are some file manipulation tasks in PC-DOS and MacOS. 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 GOMSL notation introduced in detail later in this Guide. 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. A substantive point of this example is to illustrate how a GOMS analysis can capture an important aspect of interface "consistency."

Task Instances and the Top Level

The GOMSL models in this example actually execute specified file manipulation tasks. The *form* in which these tasks are specified is called a *task description* and a specific task specified in this form is called a *task instance*. Here are the task instances used in this example, chained in sequence like a linked-list, with comments:

```
// first task is to delete the file "Work/file1.txt"
Task_item: T1
```

```
  Name is First.
  Type is delete_file.
  Filename is "file1.txt".
  Enclosing_directory is "Work".
  Next is T2.

// next task is to move the file "Work/file2.txt" to the directory "Documents"
Task_item: T2
  Name is T2.
  Type is move_file.
  Filename is "file2.txt".
  Enclosing_directory is "Work".
  Destination_directory is "Documents".
  Next is T3.

// next task is to delete the "Temp" directory
Task_item: T3
  Name is T3.
  Type is delete_directory.
  Directory_name is "Temp".
  Next is T4.

// last task is to move the "Work" directory into the "Documents" directory
Task_item: T4
  Name is T4.
  Type is move_directory.
  Directory_name is "Work".
  Destination_directory is "Documents".
  Next is None.
```

Both systems also will have the same top-level method, which is to simply get each task item in sequence, make it the current task, and then perform it:

```
Method_for_goal: perform disk_maintenance
  Step 1. Store First under <current_task_name>.
  Step 2. Decide: If <current_task_name> is None,
          Then Delete <current_task>; Delete <current_task_name>;
          Return_with_goal_accomplished.
  Step 3. Get_task_item_whose Name is <current_task_name>
          and_store_under <current_task>.
  Step 3. Accomplish_goal: perform disk_task.
  Step 4. Store Next of <current_task> under <current_task_name>.
  Step 5. Goto 2.
```

GOMSL Methods for MacOS

In the MacOS interface, as in most other GUI OS interfaces, no distinction is made between manipulating a file and manipulating a directory. Thus the selection rule for accomplishing the the perform disk_task goal can simply invoke either a move or delete method:

```
Selection_rules_for_goal: perform disk_task
  If Type of <current_task> is delete_file,
    Then Accomplish_goal: delete object using
          Filename of <current_task>.
  If Type of <current_task> is move_file,
    Then Accomplish_goal: move object using
```

```
            Filename of <current_task>, and
            Destination_directory of <current_task>.
If Type of <current_task> is delete_directory,
    Then Accomplish_goal: delete object using
            Directory_name of <current_task>.
If Type of <current_task> is move_directory,
    Then Accomplish_goal: move object using
            Directory_name of <current_task>, and
            Destination_directory of <current_task>.
  Return_with_goal_accomplished.

Method_for_goal: delete object using <object_name>
  Step 1. Accomplish_goal: drag object using <object_name>, and Trash.
  Step 2. Return_with_goal_accomplished.

Method_for_goal: move object using <object_name>, and <destination_directory>
  Step 1. Accomplish_goal: drag object using
            <object_name>, and <destination_directory>.
  Step 2. Return_with_goal_accomplished.
```

The methods for deleting and moving an object take one or two parameters; for example, the name of the object to be deleted or moved. In addition to the moving and deleting methods, there is a general submethod for the drag operation; this is the basic method used in most of the MacOS file manipulations. It is called like a subroutine by the above methods:

```
Method_for_goal: drag object using <object>, <destination>
  Step 1. Look_for_object_whose Label is <object>
            and_store_under <target>.
  Step 2. Point_to <target>.
  Step 3. Hold_down Mouse_Button.
  Step 4. Look_for_object_whose Label is <destination>
            and_store_under <target>.
  Step 5. Point_to <target>.
  Step 6. Verify "icon is at destination".
  Step 7. Release Mouse_Button.
  Step 8. Delete <target>; Return_with_goal_accomplished.
```

Finally, since in this interface, visual objects are being presented and manipulated, we must supply the required information about their visual locations and appearance. The most elaborate way to do this is with a simulated device that generates and updates the visual information about the objects. However, for simplicity, we can simply declare the existence of "dummy" visual objects with appropriate Labels as part of the GOMS model, as in:

```
Visual_item: Trash_can_icon
  Label is Trash.
Visual_item: file1_icon
  Label is "file1.txt".
Visual_item: file2_icon
  Label is "file2.txt".
Visual_item: Documents_icon
  Label is "Documents".
Visual_item: Work_icon
  Label is "Work".
Visual_item: Temp_icon
  Label is "Temp".
```

More properties could be supplied if needed by the methods. But if the visual appearance of the object needs to dynamically change in response to the simulated user's activity, a simulated device needs to be coded that will

update the properties in response to user input.

GOMS Model for PC-DOS

The PC-DOS model for the same set of tasks involve a large number of specific methods, and some of the user goals, such as moving a file, are accomplished by calling submethods. 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, and this is reflected in how the top-level selection rule maps our for sample task types to separate goals. Each of these methods calls a general submethod for entering and executing a specified command, and supplying a series of parameters. In GOMSL, `Nil` is a symbol used to represent an nil or null value.

```
Selection_rules_for_goal: perform disk_task
  If Type of <current_task> is delete_file,
    Then Accomplish_goal: delete file.
  If Type of <current_task> is move_file,
    Then Accomplish_goal: move file.
  If Type of <current_task> is delete_directory,
    Then Accomplish_goal: delete directory.
  If Type of <current_task> is move_directory,
    Then Accomplish_goal: move directory.
  Return_with_goal_accomplished.

Method_for_goal: delete file
Step 1. Accomplish_goal: perform command using
           "ERASE",
           Enclosing_directory of <current_task>, Filename of <current_task>,
           Nil, and Nil.
  Step 2. Return_with_goal_accomplished.

Method_for_goal: move file
  Step 1. Accomplish_goal: copy file.
  Step 2. Accomplish_goal: delete file.
  Step 3. Return_with_goal_accomplished.

Method_for_goal: copy file
  Step 1. Accomplish_goal: perform command using
           "COPY",
           Enclosing_directory of <current_task>, Filename of <current_task>,
           Destination_directory of <current_task>,
           Filename of <current_task>.
  Step 2. Return_with_goal_accomplished.

Method_for_goal: delete directory
  Step 1. Accomplish_goal: delete all_files_in_directory.
  Step 2. Accomplish_goal: remove directory.
  Step 3. Return_with_goal_accomplished.

Method_for_goal: delete all_files_in_directory
  Step 1. Accomplish_goal: perform command using
           "ERASE",
           Directory_name of <current_task>, "*.*" ,
           Nil, Nil.
Step 2. Return_with_goal_accomplished.

Method_for_goal: remove directory
```

```
    Step 1. Accomplish_goal: perform command using
             "RMDIR",
             Directory_name of <current_task>, Nil,
             Nil, Nil.
  Step 2. Return_with_goal_accomplished.

  Method_for_goal: move directory
    Step 1. Accomplish_goal: copy directory.
    Step 2. Accomplish_goal: delete directory.
    Step 3. Return_with_goal_accomplished.

  Method_for_goal: copy directory
    Step 1. Accomplish_goal: create directory.
    Step 2. Accomplish_goal: copy all_files_in_directory.
    Step 3. Return_with_goal_accomplished.

  Method_for_goal: create directory
    Step 1. Accomplish_goal: perform command using
             "MKDIR",
             Destination_directory of <current_task>, Nil, Nil, Nil.
    Step 2. Return_with_goal_accomplished.

  Method_for_goal: copy all_files_in_directory
    Step 1. Accomplish_goal: perform command using
             "COPY",
             Directory_name of <current_task>, "*.*" ,
             Destination_directory of <current_task>, "*.*" .
    Step 2. 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: perform command using
             <command>,
             <directory1>, <filename1>,
             <directory2>, and <filename2>
    Step 1. Type_in <command>.
    Step 2. Accomplish_goal: enter filespec using
             <directory1>, and <filename1>.
    Step 3. Decide: If <directory2> is Nil, and <filename2> is Nil,
             Then Goto 5.
    Step 4. Accomplish_goal: enter filespec using
             <directory2>, and <filename2>.
    Step 5. Verify "command is correctly entered".
    Step 6. Keystroke CR.
    Step 7. Return_with_goal_accomplished.

  Method_for_goal: enter filespec using <directory>, and <filename>
    Step 1. Keystroke SPACE.
    Step 2. Decide: If <directory> is Nil, Then Goto 5.
    Step 3. Type_in <directory>.
    Step 5. Decide: If <filename> is Nil, Then Return_with_goal_accomplished.
    Step 4. Decide: If <directory> is_not Nil, Then Keystroke "\".
    Step 6. Type_in <filename>.
    Step 7. Return_with_goal_accomplished.
```

Comparison of MacOS and PC-DOS GOMS Models

Clearly there is a substantial difference in the number and length of methods between these two systems. The GLEAN tool provides some quantitative metrics in terms of *units* of procedural knowledge which will be explained later; for now, a unit is roughly equivalent to a statement or step in GOMSL. For MacOS, the following GLEAN output shows the number of units of procedural knowledge to be learned in the the individual methods and the total:

```
Mac_file_example.gomsl
LEARNING ANALYSIS
--------------------------------------------------------------------------------
Method             Units:  number   learned  to learn   Source Method
--------------------------------------------------------------------------------
perform disk_maintenance     7         0         7
perform disk_task            6         0         6
delete  object               3         0         3
move    object               3         0         3
drag    object               9         0         9
--------------------------------------------------------------------------------
Totals:                     28         0        28
```

The top-level method, `perform disk_maintenance`, and the selection rule set, `perform disk_task`, include a total of 13 units. The methods for the individual task goals include only 3 methods containing a total of only 15 units, for a total of 28 units to be learned.

For PC-DOS, the learning analysis shows many more methods, and also some overlap between a couple of methods:

```
Dos_file_example.gomsl
LEARNING ANALYSIS
--------------------------------------------------------------------------------------
Method                   Units:  number   learned  to learn   Source Method
--------------------------------------------------------------------------------------
perform disk_maintenance           7         0         7
perform disk_task                  6         0         6
delete  file                       3         0         3
move    file                       4         0         4
copy    file                       3         0         3
delete  directory                  4         0         4
delete  all_files_in_directory     3         0         3
remove  directory                  3         0         3
move    directory                  4         4         0   move    file
copy    directory                  4         0         4
create  directory                  3         0         3
copy    all_files_in_directory     3         0         3
perform command                    8         0         8
enter   filespec                   8         0         8
--------------------------------------------------------------------------------------
Totals:                           63         4        59
```

The top-level method, `perform disk_maintenance`, and the selection rule set, `perform disk_task`, are essentially the same as the MacOS example, and require a total of 13 units. However, an additional 12 methods with a total of 50 units are needed. But one of these methods, `move directory`, is *similar* to a previously learned method, `move file,` and so, according to rules described below, this method does not have to be learned, resulting in a total of 59 units of procedural knowledge that must be learned to accomplish these goals in PC-DOS.

Thus we have a clear characterization of the extreme simplicity and consistency of MacOS 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 this property of *method consistency*. Furthermore, the difference can be quantified as 28 units of procedural knowledge versus 59.

The Guide describes below how these values can be used to derive predictions of learning 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 approximately linear with the number of units to be learned, so the MacOS methods are predicted to be far faster to learn than the PC-DOS methods.

The execution time can be computed by generating the trace through the methods required to perform a specified set of benchmark tasks. For example, for the task instances specified above, the GLEAN tool runs the MacOS methods and produces the following output:

```
Mac_file_example.gomsl: Method Execution Profile
Total Execution Time:     33.300 seconds
Freq.    Subtotal  Avg. Time  % of Total   Method for goal
   2      13.600      6.800       40.84  delete object
   4      26.400      6.600       79.28  drag object
   2      13.200      6.600       39.64  move object
   1      33.300     33.300      100.00  perform disk_maintenance
   4      27.200      6.800       81.68  perform disk_task
```

The table shows the total execution time for the set of four benchmark tasks, and an execution time profile for the set of methods. The profile includes the number of time each method was executed, the total time spent in that method, the average time in the method, and the percentage of the total time spent in each method. These percentages do not sum to 100% because of the hierarchical structure of the methods. For example, 100% of the time was spent in the top-level method, `perform disk_maintenance`, and most of that time was spent under the control of the selection rule set, `perform disk_task`, that picks out the particular goal; 41% of the time was spent deleting objects, and 40% on moving objects (half of the tasks were deletions; the other half were moves). Since both deleting and moving were done by dragging, 80% of the total time was actually spend in the drag method.

Running the PC-DOS methods on the same four benchmark tasks, the GLEAN tool produces the following output:

```
Dos_file_example.gomsl: Method Execution Profile
Total Execution Time:     76.100 seconds
Freq.    Subtotal  Avg. Time  % of Total   Method for goal
   1      10.400     10.400       13.67  copy all_files_in_directory
   1      16.900     16.900       22.21  copy directory
   1      13.800     13.800       18.13  copy file
   1       6.350      6.350        8.34  create directory
   2      12.500      6.250       16.43  delete all_files_in_directory
   2      22.700     11.350       29.83  delete directory
   2      15.900      7.950       20.89  delete file
  11      39.550      3.595       51.97  enter filespec
   1      28.400     28.400       37.32  move directory
   1      21.900     21.900       28.78  move file
   9      67.950      7.550       89.29  perform command
   1      76.100     76.100      100.00  perform disk_maintenance
   4      70.000     17.500       91.98  perform disk_task
   2       9.900      4.950       13.01  remove directory
```

Again, the percentages do not sum to 100% because of the hierarchical structure of the methods. As expected from the benchmark tasks, roughly a quarter of the time was spent in the individual methods for moving/copying files/directories, but actually the move directory task was relatively time-consuming at 37% of the total (an average of about 28 s), while deleting a file was also surprising large at 21% of the total, even though it required only about 8 s on the average, because it was executed twice - once for deleting a file, and once for moving a file. About 90% of the time was spent entering commands, which makes sense, and most of that time was spent entering the file

specifications - which had to be done a total of 11 times in these benchmarks.

The MacOS methods, even though they use relatively slow visual searches and mouse point operations, still took only about half the time of the PC-DOS methods (about 33 s versus 76 s), showing an execution speed advantage for MacOS on these benchmark tasks and these methods (other comparisons might show how typing PC-DOS pathnames might be faster than opening multiple windows in MacOS to locate a file). The execution profile shows that a large part of the PC-DOS time was spend in entering commands and file specifications.

1.4 Organization of this Guide

This presentation supersedes earlier presentations of NGOMSL, GOMSL, and GLEAN (Kieras, 1988, 1997a; Kieras, Wood, Abotel, & Hornof, 1995). The remainder of this Guide is organized as follows: Section 2 defines the GOMSL notation for GOMS models and their GLEAN implementation. Section 3 discusses some of the general issues that underlie GOMS modeling. 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. Section 5 provides an introduction to using the GLEAN tool.

2. GOMSL: A NOTATION FOR GOMS MODELS

This section defines presents a notation system, GOMSL (**GOMS L**anguage), which is an executable representation for GOMS models that is based on the earlier NGOMSL (Kieras, 1988, 1997a). GOMSL is an attempt to define a language that will allow GOMS models to be written and executed, but that is also easy to read rather than cryptic and abbreviated. However, it is important to keep in mind that GOMSL 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 GOMSL is supposed to represent something like "the programming language of the mind," as absurd as this sounds. The idea is that GOMSL programs have properties that are related in straightforward ways to both data on human performance and theoretical ideas in cognitive psychology. If GOMSL is clumsy and limited as a computer language, it is because humans have a different architecture than computers. Thus, for example, GOMSL 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 GOMSL program. In this document, GOMSL expressions are shown in `this typeface`.

2.1 Data Representation and Storage

Object-Property-Value Representation

Values of data in GOMSL are represented with *symbols*, which are either *identifiers* , *strings*, or *numbers*. Identifiers are case-sensitive character strings beginning with a alphabetic character and containing any number of alphabetic characters, digits, the hyphen character '-', or the underscore character '_'. A string is the usual sequence of characters enclosed in double quotes and can contain embedded blanks or special characters. Numbers at this time must be enclosed in double-quotes, although they are represented internally as double-precision floating point value. All of the following are examples of symbols, with the first two being identifiers, the second two strings, and the last two are numbers:

```
NCC1701
skilled_trade
"Favorite phrase"
"What's up, Doc?"
"700"
"3.14"
```

There are several symbol names built into GOMSL and GLEAN; generally these start with an initial upper-case letter. There are two used pervasively, and so deserve special mention. `Absent` means that a sought-for object or item is not present. `Nil` is used to indicate a value that that is nothing, similar to the constant `nil` in LISP.

The basic data representation in GOMSL consists of *objects* with *properties* and *values*. Each object has a symbolic name and a list of properties, each of which has an associated value. The object name, property, and value are symbols. This representation is used in several places: For example, long-term memory is represented as a collection of objects, or items, each of which has a symbolic name and a set of property-value pairs. For example, the fact that "plumber" is a skilled trade and has high income might be represented as follows:

```
LTM_Item: Plumber.
  Kind is skilled_trade.
  Income is high.
```

In this example, "Plumber" is an object in LTM that has a "Kind" property whose value is "skilled_trade" and an "Income" property whose value is "high."

Another example is declarative knowledge of an interface as a collection of facts about the Cut command in a typical text editor:

```
LTM_Item: Cut_Command.
  Containing_Menu is Edit.
  Menu_Item_Label is Cut
  Accelerator_Key is Command-X.
```

The "cut" command is described as an object whose properties and values specify which menu it is contained in, the actual label used for it in the menu, and the corresponding accelerator (short-cut) key. Likewise, a task instance is described as a collection of objects each of which has properties and values. There are operators for accessing or retrieving visual objects, task or long-term memory items, and then accessing their individual properties and values.

As mentioned above, the value in a property-value pair can also be a real number; at this time, such a value is specified like a string, but represented internally as a double-precision floating point number. For example, if our plumber has an hourly fee of $55.60/hr, we might specify it as:

```
  Hourly_rate is "55.60".
```

Working Memory

Working memory in GOMSL consists of two kinds of information: one is a *tag store* (Kieras, Meyer, Mueller, & Seymour, 1999), which represents an elementary form of working memory. The other kind is the *object store* which holds information about an object that has been brought into "focus", that is, placed in working memory and whose property values are thus immediately available in the form of a *property-of-object* construction.

Tag store. The working memory tag store consists of a collection of symbolic values stored under a symbolic name or *tag*. Tags are expressed as identifiers enclosed in angle brackets. In many cases, the use of tags corresponds to traditional concepts of verbal working memory; syntactically, they roughly resemble variables in a traditional programming language. At execution time, if a tag appears in an operator argument, it is replaced with the value currently stored under the tag. An elementary example:

```
Step 1. Store "foo.txt" under <filename>.
Step 2. Type_in <filename>.
```

In Step 1, the `Store` operator is used to place the string "foo.txt" into working memory under the tag `<filename>`. In Step 2, before the Type_in operator is executed, the value stored under the tag is retrieved from working memory, and this becomes the parameter for the operator. So Step 2 results in the simulated human typing the string "foo.txt".

Object store. There is also a working memory store for visual input, task information, and long-term memory retrievals. All three of these capture the common feature that gaining access to an object or item will be time consuming, but once it has been located or retrieved, further details of the object or the item can then be immediately used by specifying the desired property of the object with a *property of object* construction. So the operation of bringing an object or item into focus is time-consuming, but then all of its properties are available in working memory. But if the "focus" is changed to a different object or item, the information is no longer available, and a time-consuming operation is required to bring it back into focus.

This mechanism represents in a simple way the performance constraints involved in many forms of working memory and visual attention. For example, suppose a user must search the display for a text field with a certain label; once it is found, the eye is now fixated on the text field, and so a decision can then be quickly made about its color. If the eye is then moved to another object, the immediate access to the first object is lost, and the second object is then in focus. Consider the following example, in which the *property of object* construction in Step 2 works, but the same construction in Step 4 is invalid:

```
Step 1. Look_for_object_whose Label is Result,
  and Type is text_field and_store_under <result_field>.
Step 2. Decide: If color of <result_field> is red, Then Keystroke X.
```

```
Step 3. Look_for_object_whose color is blue and_store_under <button>.
Step 4. Decide: If color of <result_field> is red, then Keystroke X.
```

In Step 1, the simulated user searches the screen for a visual object whose "Label" property has the value "Result" and whose "Type" property has the value "text_field"; if found, the symbolic name of the object is stored in the tag store under the tag "`<result_field>`". This visual search operation takes an assumed standard time to execute. The visual object is now "in focus", available in visual working memory. So in Step 2, the color of this object is accessed with an expression of the form *property of object* and compared to a value. Since the `<result_field>` object is in focus in working memory, its color is assumed to be immediately available, and so the additional time cost of accessing the color property is assumed to be zero, and the time for testing the color property is bundled into the step execution time. Step 3 starts a visual search for a blue object; this displaces the `<result_field>` object from focus, and replaces it with the `<button>` object. Thus Step 4 produces a run-time error because it attempts to access the property of an object that is no longer in focus.

Each of the three working memory stores for visual, task, and long-term memory retrievals is an independent partition of working memory, meaning that simultaneously a visual object can be in focus visually, a task item can be available, and the results of a long-term memory retrieval are in working memory. Details of the search and retrieval operators for each object store are presented below.

This analysis is a simplification of the very complex ways in which working memory information is accessed and stored during a task (see Kieras, Meyer, Mueller, & Seymour, 1999, for more discussion).

2.2 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 a pair of identifiers, which by convention are chosen to be an action-object pair in the form: ***`verb noun`***, such as `delete word`, or `move-by-find-function cursor`. Either the noun or the verb can be complicated if necessary to distinguish between methods (see below on selection rules). Any parameters involved that modify or specify a goal, such as where a to-be-deleted word is located, are represented in the task description, and made available when the method is invoked (see below).

2.3 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 `Keystroke ENTER`. 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 chosen by the analyst (John & Kieras, 1996b).

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 constructing a GOMS model 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 operators* 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 humans. These include perceptual operators, which read text from a screen, scan the screen to locate the cursor, or input a piece of speech, and motor operators, such as pressing a key, or speaking a phrase. The analyst usually chooses the external operators depending on the system or tasks, such as whether there is there a mouse on the machine. Some external operators, such as finding something on the display has some overt components such as eye movements, but there are also considerable mental components of the activity; these are treated as external operators, but with time estimates based on mental operators.

Mental operators. Mental operators are the internal actions performed by the user; most 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, storing an item in Working Memory (WM), retrieving information from Long-Term Memory (LTM), determining details of the next task to be performed, or setting up a goal to be accomplished.

Other mental operators are defined by the analyst to represent complex mental activities (see below), normally as a placeholder for a complex activity that cannot be analyzed further. A common such *analyst-defined* mental operator is verifying that a typed-in command is correct before hitting the `ENTER` key; another example would be a stand-in for an activity that will not be analyzed, such as `LOG-INTO-SYSTEM`.

Inter- vs intrastep operators. As explained in more detail later, based on the underlying production system model for human cognition, each step in a GOMS model takes a certain time to execute, estimated at 50 ms. Most of the built-in mental operators are all executed during this fixed step execution time, and so are termed *intrastep* operators. However, substantially longer execution times are required for external operators, such as pointing to the target with a mouse, or certain built-in mental operators such as searching long-term memory. In the GLEAN architecture, these slow operators must be completed before execution of the next step in the thread is begun. Thus, these are *interstep* operators because their execution time occurs between the steps, and so governs when the next step is started.

Primitive and high-level operators. A particular GOMS model 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.

However, the GLEAN architecture only incorporates a specific set of operators whose execution time characteristics are known. One of these is a generic "mental" operator, which often suffices as a placeholder for complex mental processes. So if the analyst chooses to include non-standard operators that he or she has created for a particular problem, the current GLEAN tool will not be able to parse the GOMS model, and the execution time cannot be predicted by the GLEAN simulation. As GLEAN develops, additional operators can be easily included. In addition, future versions of GLEAN will include a facility for specifying an arbitrary operator name and associating it with a fixed time.

Standard Primitive External Operators

Listed here are the primitive motor and perceptual operators whose definitions and execution times are based on the physical and some of the mental operators used in the Keystroke-Level Model(Card, Moran, & Newell, 1983; John & Kieras, 1996a, b). These are all interstep operators. Based on the simplifying logic in the Keystroke-Level Model, operators for complex mental activities are assumed to take a constant amount of time, approximated with the value of 1200 ms, based on results in Olson & Olson (1989).

Each operator keyword is listed in `this typeface`; parameters for operators are shown in *this typeface*. Unless otherwise stated, an operator parameter can be either a symbol, a tag, or a *property of object* construction. An expression enclosed in angle-brackets, as in <*tag*> is a tag name parameter; its resemblance to a variable item in BNF notation is a deliberate contribution to the readability of GOMSL, but for purposes of defining the GOMSL notation itself, it is a specific type of expression, similar to a variable in a standard programming language. A description of the operator and its execution time for each operator is given.

Manual Operators

`Keystroke` *key_name*
Strike a key on a keyboard. If the keyname is a string, only the first character is used. 280 ms

`Type_in` *string_of_characters*
Do a series of Keystrokes, one for each character in the supplied string. 280 ms/character

`Click` *mouse_button*
The designated mouse button is pressed and released. 200 ms

`Double_click` *mouse_button*
Two Clicks are executed. 400 ms

`Hold_down` *mouse button*
Press and continue to press the mouse button. 100 ms

`Release` *mouse_button*
Release the mouse button. 100 ms

`Point_to` *target_object*
The mouse cursor is moved to the target object on the screen. Time required is determined by Welford form of Fitts' Law with a minimum of 100 ms if object location and sizes are specified; defaults to 1100 ms if not.

`Home_to` *destination*
Move the right hand to the destination. The initial location of the right and left hands is the keyboard. Possible destinations are `keyboard` and `mouse.` 400 ms

All of the manual operators automatically execute a `Home_to` operator if necessary to simulate the hand being moved between the mouse and keyboard.

Vocal Operator

`Speak` *utterance*
The utterance is output as a symbol whose value is equal to the utterance symbol (e.g. a string). The time to speak the utterance is 150 ms/syllable, where the gap between words is also counted as a syllable. Currently the syllables counting algorithm is a simple approximation, which should be improved in the future.

Visual Operators

Look_for_object_whose *property is value, ...* and_store_under *<tag>*
This is a mental operator that searches visual working memory, (essentially the visual space) for an object whose specified properties have the specified values, and stores its symbolic name in working memory under the label *<tag>* where it is now in focus. If no object matching the specification is present, the result is the symbol Absent, which may be tested for subsequently. Time required is 1200 ms, after which additional information about the object is available immediately. Putting a different object in focus will result in the previous object's properties being no longer available. If the visual object disappears (e.g. it is taken off of the display screen), then it will be removed from visual working memory, and the object information is no longer available. Static visual objects and their properties can be defined as part of the GOMS model.

Look_at *object_name*
The object_name is supposed to be the name of an existing visual object has been identified by a previous Look_for operator, or is known as a visual property of another visual object. For example, object1 might have the Above property for object2; if object1 is in focus, then Above of object1 will identify object2. The time required is assumed to faster than a Look_for, because no visual search is required, and is set at 200 ms.

Wait_for_visual_object_whose *property* is *value*, ... and_store_under *<tag>*
This mental operator is equivalent to the Look_for_object operator, but with the exception that if the specified object is not already present, execution of the method is suspended until it is. If the object is already present, then if it is in focus, the operator takes no time; if it is not in focus, then the standard 1200 ms mental operator time is required. If the operator waits for the matching object to become present, then no additional time is required if it was already in focus (e.g., the user is watching an indicator and waits for it to change color). If it was not in focus, then an additional 1200 ms mental operator time is required, representing the search and recognition process involved.

Auditory Operator

Wait_for_auditory_object_whose *property* is *value*, ... and_store_under *<tag>*
A mental operator that waits until auditory input appears that matches the specification; it is similar to the corresponding visual operator, but has some important differences since the auditory processor is involved. An input contains a symbol corresponding to the input sound; this auditory stimulus is assigned an object name consisting of a unique number and saved in auditory working memory. Additional properties and values specified in the auditory encodings (see below) is added to the object; these properties may be used to specify the desired object. After a standard time delay representing auditory working memory decay time (currently 1000 ms), this object is deleted and the auditory stimulus information is no longer available. If the specified object is present, a total of 150 ms is required, representing the auditory recognition processing. Otherwise, the operator waits until the specified object becomes present, and the same 150 ms delay applies. For simplicity, it is assumed that all auditory objects in auditory working memory are simultaneously in focus.

Standard Primitive Mental Operators

Below follows a brief description of the GOMSL primitive mental operators; examples of their use appear later. These are all intrastep operators.

Basic flow of control. A submethod is invoked by asserting that its goal should be accomplished:

```
Accomplish_goal: goal
AG: goal (abbreviation)
Accomplish_goal: goal using pseudoargument tag list
```

This is analogous to an ordinary CALL statement; control passes to the method for the goal, and execution started on the next step, and returns here when the goal has been accomplished. The pseudoargument tag list is described in a later section below. The operator:

```
Return_with_goal_accomplished
RGA (abbreviation)
```

is analogous to an ordinary RETURN statement. A method normally must contain at least one of these.

There is a branching operator:

```
Goto step_label
```

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

A decision is represented by a step containing a `Decide` operator; a step may contain only one `Decide` operator and no other operators. The `Decide` operator contains one or more `If-Then` conditionals and at most one `Else` form. The conditionals are separated by semicolons:

```
Decide: conditional
Decide: conditional; conditional; ... else-form
```

The `If` part of a conditional can have one or more predicates. The `Then` part or an else-form has one or more operators:

```
If predicates Then operators
Else operators
```

The predicates consist of one or more predicates, separated by commas or comma-and. If all of the predicates are true, then the operators are executed, and no further conditionals are evaluated. An else-form may appear only at the end; its operators are executed only if all of the preceding `If` conditions have all failed to be satisfied. Normally, `Decide` operators are written so that the if-else combinations are mutually exclusive. Note that nesting of conditionals is not allowed. Here are three examples of `Decide` operators:

```
Step 3. Decide: If <id_letter> is "A" Then Goto 4.

Step 8. Decide:
  If appearance of <current_thing> is "super duper", and
          size of <current_person> is large,
    Then Type_in string of <current_task>;
  If appearance of <current_thing thing> is Absent, Then RGA;
  Else Goto 2.

Step 5. Decide:
  If <button_label> is ACCEPT Then Keystroke K1;
  If <button_label> is REJECT, Then Keystroke K2;
  Else Keystroke K3.
```

If there are multiple `If-Then` conditionals, as in the second and third example above, the conditions are supposed to be mutually exclusive, so that only one condition can match, and the order in which the `If-Thens` are listed is supposed to be irrelevant. However, in practice, the conditionals are evaluated in order, and evaluation stops at the first conditional whose condition is true.

The `Decide` operator is used for making a simple decision that governs the flow of control within a method. It is not supposed to be used to implement GOMS selection rules, which have their own construct (see below). The `If` clause typically contains predicates that test some state of the environment or contents of WM. 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.

The following predicates are currently implemented:

```
is   is_equal_to (synonyms)
is_not   is_not_equal_to (synonyms)
is_greater_than
is_greater_than_or_equal_to
is_less_than
is_less_than_or_equal_to
```

These predicates are valid for either numeric or symbol values. If the compared values are both numeric, then a comparison between the numeric values is performed. Otherwise both values are treated as symbolic, and the character string representations of the two are compared in lexicographic order - if necessary, a number is converted to a standard string representation for this purpose.

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. WM contents are normally stored and retrieved using their *tags*, the symbolic name for the value being stored in working memory. These tags are used in a way that is somewhat analogous to variables in a conventional programming language.

`Store value under <tag>`
The value is stored in working memory under the label <tag>.

`Delete <tag>`
The value stored under the label <tag> is deleted from working memory.

`Recall_LTM_item_whose property is value, ... and_store_under <tag>`
Searches long-term memory for an item whose specified properties have the specified values, and stores its symbolic name in working memory under the label <tag> where it is now in focus. Time required is 1200 ms, after which additional information is available immediately. Putting a different task item in focus will result in the previous item's properties being no longer available.

Consistent with the theoretical concept that working memory is a fast scratch-pad sort of system, and how it is used in the production system models, the execution time for the `Store` and `Delete` operators is bundled into the time to execute the step; thus they are intrastep operators. The `Store` operator is used to load a value into working memory. The `Delete` operator is more frequently used to eliminate working memory items that are no longer needed. Although this deliberate forgetting might seem counter-intuitive, it is a real phenomenon; see Bjork (1972). In the GLEAN architecture, working memory information does not "decay" and so there is no built-in limit to how much information can be stored in working memory. This reflects the fact that despite the considerable research on working memory, there is not a theoretical consensus on what working memory limits apply during the execution of procedural tasks (see Kieras, Meyer, Mueller, & Seymour, 1999). So rather than set an arbitrary limit on when working memory overload would occur, the analyst can identify memory overload problems examining how many items are required in WM during task execution; GLEAN can provide this information (see below, on mental workload).

There is only a recall operator for LTM, because in the tasks typically modeled with GOMS, long-term learning and forgetting are not involved. The `Recall_LTM_item` operator is an interstep operator that takes a standard mental operator execution time, but once the information has been placed in focus in WM, additional information about the item can be used immediately with the `x_of_y` argument form.

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 to Keystroke-Level Model *Mental* operators, and so can be approximated with as standard interstep mental operators that require 1.2 sec. The Verify and Think_of operators are intended for this situation; the analyst simply documents the assumed mental process in the description argument of the operator. The GOMSL language does not provide any way to implementing genuine new operators that perform novel computations - such operators have to be added to the GLEAN architecture itself.

Verify *description*
Think_of *description*
These operators simply introduce a time delay to represent when the user must pause and think of something about the task; the actual results of this thinking are specified elsewhere, such as the task description. The description string serves as documentation, nothing more. Each operator requires 1200 ms.

Task Memory Access Operators

The task store, or task memory, is normally used to represent the information available to the user about the tasks to be executed and their details. The basic idea is that the analyst does not wish to specify the actual psychological or physical mechanism for the task memory information and its maintenance, so instead these operators are used as approximate representations for the actual process. The task memory can be used in two ways: One is as a passive store of information, such as a reliable memory for the tasks to be done, or as an external source, such as a marked-up manuscript or a set of notes (see the "yellow pad" heuristic below). A second way is as an interactive store that the human can modify relatively slowly during task execution, such as modifying the information on a written set of notes, or changing information in some kind of task-specific working memory. While originally developed to support modeling of complex tasks (Kieras & Santoro, 2004), task memory closely resembles the concept of *long-term working memory* proposed by Ericsson & Kintsch (1995; Ericsson & Delaney, 1999), which is basically a medium-access-speed large-capacity working memory based on a stable organizational structure that skilled performers develop to use in a specific task domain. Further research is needed to bring these ideas into alignment. But for purposes of modeling, if the simulated human is assumed to be highly skilled at the modeled task, and the information needed to perform the task can be organized using a stable pattern, then it would be appropriate to use this task memory to represent it. An especially useful form of information in task memory is a list of items; operators are provided to construct and maintain lists of items. The task information itself consists of a set of Task-items, presented below, that together comprise a collection of objects and properties.

General-Purpose Task Memory Operators

Get_task_item_whose *property* is *value*, ... and_store_under *<tag>*
The Task store is searched for an object matching the property-value pairs, and the name of the first one found is placed in <tag>. If the specified object is not found in the task description, then the resulting symbol is Absent, and this value can be subsequently tested for. The task item is now in focus, and additional properties of the item are now available. The time required is 1200 ms. Putting a different task item in focus will result in the previous item's properties being no longer available.

Write_task_item_whose *property* is *value*, ... and_store_under *<tag>*
An object is created in the task store whose name is of the form Tobj*nnn* and whose properties and values are those specified. The new task item is now in focus.

Update_for_task_item *item_name* that *property* is *value*, ...
Sets the properties of the named task item to the supplied properties and values. It is an error if the named item does not exist in the Task store already. The updated task item is now in focus.

Erase_task_item_property *item_name property*
Removes the named property from the named task item.

`Erase_task_item` *`item_name`*
Removes the named object from the Task store.

Task List Operators

These operators are used to keep Task store objects in linked lists that can be accessed in the usual ways linked-lists can be accessed. There can be multiple lists. Two kinds of objects are involved:

> 1. A *List* object represents the list as a whole. Its `Label` property is the name of the list, and it has `First` and `Last` properties that name the first and last nodes in the list. It is created by the `Create_task_list` operator. The actual name of the List object is created automatically, like the other Task objects, and has a name in the form `Tobj`*`nnn`*.
>
> 2. A *List Item* object represents an item in the list. The name of a list item object is referred to as the *item name* in what follows. It is created by the normal Task operators, such as `Write_task_item_whose`, and thus can have various properties and values. However, the list operators assign three additional properties to make this Task object a member of the linked list. The `In_List` property names the list the node belongs to (the same as the List Header `Label`). The `Next` and `Previous` properties name the nodes coming before or coming after, or `Absent` if there are none. The modeler should take care not to change these properties, or the list structure might become invalid.

The normal pattern of usage is to first find an item in the list and get its item name, and then using ordinary operators, you can test or change the values of whatever properties of the list item you want. If a Task list operator is used to erase or remove an item from a list, that item object is normally not erased from the task store - rather, all of the List specific properties are removed. By displaying the contents of the task store, the modeler can examine the state of all the currently existing task lists.

Except where an list or item is being created, or noted otherwise, it is an error if there is no list of the supplied name, or if there is no item with the supplied name.

Operating on a Whole List

`Create_task_list` *`list_name`*
Creates a new and empty list of the supplied name: a List object with `Label` of list_name is created in the task store. To simplify model restarts, this operator does nothing if a list of that name is already present.

`Erase_task_list` *`list_name`*
Empties the named task list, leaving it in the same state as if it had just been created; as a convenience, unlike the list item removing operators, this operator also removes the list item objects from the Task store.

`Get_size_of_task_list` *`list_name`* `and_store_under` *`<tag>`*
The number of items (list item objects) in the named list are counted and stored in the tag.

Adding New Items to the List

The following operators place an already existing task object in a list by setting the `In_list`, `Previous`, and `Next` properties appropriately; any previous values of these properties are overwritten. The list object is also updated.

`Prepend_item` *`item_name`* `to_task_list` *`list_name`*

Append_item *item_name* to_task_list *list_name*
The specified task object becomes a list item by being added to the begining or end of the named task list. I.e., the `In_list`, `Previous` and `Next` properties of the task object are set to its being the first or last item in the list, and the List object and any previous first or last list item are updated.

Insert_item *item_name1* before *item_name2* in_task_list *list_name*
The first item is added to the list before the second item.

Finding Items in the List

The generic `Get_task_item_whose ...` operator can be used to find any desired task object in the list by specifying the `In_list` property and any other desired properties, including the `Previous` and `Next` properties. The list-specific operators described here automate some of these property specifications and are more expressive. These operators search a named list for an item, and put the item name in the supplied tag. If a matching item is not found, `Absent` is stored in the tag.

Find_item_whose *property* is *value*... in_task_list *list_name* and_store_under *<tag>*
The named list is searched for the first item whose specified properties match the specified values, and the item name of the item is stored in the tag.

Find_first_item_in_task_list *list_name* and_store_under *<tag>*
Find_last_item_in_task_list *list_name* and_store_under *<tag>*
The first or last item in the named list is located, and the name of the item is stored in the tag.

Find_item_before *item_name* and_store_under *<tag>*
Find_item_after *item_name* and_store_under *<tag>*
The name of the item that comes before/after the specified item is stored under the tag.

Updating Items in the List

The regular `Update_for_task_item` should be used to change the properties for a task list item, once it has been found. Care should be taken not to change the `In_list`, `Previous`, or `Next` properties. Similarly, the `Erase_task_item_property` operator can be used to remove a single property and its value from a task object

Removing Items from the List

These operators will remove an item from the list, and modify the `Previous` and `Next` properties of the other list items and the list object to leave the rest of the list in a correct state. They do not remove the task item itself, but they will remove the `In_list`, `Previous`, and `Next` property of the item to reflect that it is no longer in a list. The regular `Erase_task_item` operator can be used to remove the item itself, with all of its remaining properties, from the Task store.

Remove_item *item_name* from_task_list *list_name*
The named task item is removed from the list, but not from the task store.

Remove_first_item_from_task_list *list_name*
Remove_last_item_from_task_list *list_name*
The first or last item in the list is removed from the list, but not from the task store.

2.4 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 analysis.

The form for a method is as follows:

```
Method_for_goal: goal
Step 1. operators.
Step 2. operators.
...
```

Abbreviations and pseudoparameters are allowed:

```
MFG: goal (abbreviation)
Method_for_goal: goal using pseudoparameter tag list
```

Steps

More than one operator can appear in a step, and at least one step in a method must contain the operator `Return_with_goal_accomplished`. A step starts with the keyword `Step`, contains an optional label, followed by period, and one or more operators separated by semicolons, with a final period:

```
Step. operator.
Step label. operator.
Step. operator; operator; operator.
Step label. operator; operator; operator.
```

The label is either a number or an identifier. The labels are ignored by GLEAN except for the `Goto` operator, which searches the method from the beginning for the first matching label; this designates the next step to be executed. Thus the labels do not have to be unique or in order. However, a run-time error occurs if a `Goto` operator does not find a matching label. Using numeric labels throughout highlights the step-by-step procedure concept of GOMS methods, but plan on renumbering the steps and altering `Gotos` to maintain a neat appearance.

Operator Restrictions

The operators are executed in the order they appear. Because some operators change the flow of control, and interstep operators suspend execution of the method for a time, there are restrictions on which and how many operators can appear in the list of operators that comprises a step, or appears following `Then` in a `Decide` conditional. These restrictions are as follows:

• If a `Decide` operator appears, it must be the only operator in the step, and the list of operators in each conditional must obey the following restrictions for operator lists.

• Only one of the following simple flow of control operators is allowed in a list of operators, and it must appear at the end of the list:

```
Goto, Accomplish_goal, Return_with_goal_accomplished
```

• Any number of WM `Store` and `Delete` operators may appear in an operator list. They are executed in the order listed, and are intrastep operators, so their execution times are bundled into the execution time for the step.

• Only one interstep operator can appear in an operator list; these are time-consuming operators whose duration is longer than a single step time, and so the next step will not be executed until this operator completes. For example, the following steps are allowed:

```
Step 1. Store "A" under <key_to_hit>; Keystroke <key_to_hit>;
         Delete <key_to_hit>; Accomplish_goal: analyze results.
Step 2. Decide: If <key_to_hit> is "A" Then Goto 4.
Step 3. Retrieve_from_LTM "key_is" of <Alpha>
         and_store_under <key_to_hit>.
```

But not:

```
Step 1. Goto Start; Keystroke DEL; Point_to <window>.
         Think_of "file to search".
```

Method Hierarchy

Methods often call submethods to accomplish goals that are subgoals. This method hierarchy takes the following form:

```
Method_for_goal: goal
  Step 1. operators.
  Step 2. <operators>
  ...
  Step i. Accomplish_goal: subgoal.
  ...
  Step m. Return_with_goal_accomplished.

Method_for_goal: subgoal
  Step 1. operators.
  Step 2. <operators>
  ...
  Step j. Accomplish_goal: sub-subgoal.
  ...
  Step n. Return_with_goal_accomplished.
  ...
```

Method Pseudoparameters

The simple tagged-value model of working memory results in WM tags being used something like variables in traditional programming languages, but because there is only one WM system containing only one set of tagged values, these "variables" are effectively global in scope. This makes it syntactically difficult to write "library" methods that represent reusable "subroutines" with "parameters." To alleviate this problem, you can specify *pseudoarguments* when you call a method with the `Accomplish_goal` operator and the corresponding *pseudoparameters* when you define a method or selection rule set. For example:

```
Step 8. Accomplish_goal: Enter Data using "Name",
         and name of <current_person>.

Method_for_goal: Enter Data using <field_name>, and <data>
  Step 1. Look_for_object_whose label is <field_name>
           and_store_under <field>.
  Step 2. Point_to <field>.
  Step 3. Click <button>.
  Step 4. Type_in <data>.
  Step 5. Delete <field>; Return_with_goal_accomplished.
```

The items in the `using` list play the role of subroutine call arguments and parameters under a call-by-value paradigm. However, they are not real arguments and parameters because a single WM system is used instead of a function call stack, and the argument and parameter names occupy the same scope; hence the *pseudo-* prefix. In the

example above, Step 8 of the calling method invokes the `Enter Data` submethod using two pseudoarguments. The values of these two are computed, and implicitly a `Store` operator is executed to store them under the pseudoparameter tags, the first value under the tag `<field_name>` and the second under the tag `<data>`. The submethod makes use of the tags in the normal way.

But when the `Enter Data` submethod returns, an implicit `Delete` operator is executed for the tags `<field_name>` and `<data>`. Note that the method is still responsible for cleaning up the "local variable" `<field>`. The tag `<button>` is like a global variable; The calling methods must have stored something under the tag `<button>` prior to calling the `Enter Data` method, and this value and tag is not altered by this method. But a submethod can "return" a value to the caller by storing something under a tag which the calling method then uses.

The pseudoargument approach is crude but a simple way to allow for an easily reusable methods library, without the theoretically questionable approach of parameter-passing and local scopes using a function-call stack that implies an unrealistic working memory mechanism. Future experience will tell whether this crude approach is adequate for actual applications of GOMSL.

2.5 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-arrow-keys cursor`. But if the place is far away, then the general goal should be accomplished by the specific one of `move-with-find-function cursor.`

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_rules_for_goal: general goal
If predicates Then Accomplish_goal: specific goal.
If predicates Then Accomplish_goal: specific goal.
...
Return_with_goal_accomplished.
```

If pseudoparameters are involved, the form is:

```
Selection_rules_for_goal: general goal using pseudoparameter tag list
If predicates
  Then Accomplish_goal: specific goal using pseudoargument list.
...
Return_with_goal_accomplished.
```

The `If-Then` conditionals are identical in syntax to those in the `Decide` operator, with the exception that the only operator that can appear in the `Then` part is the `Accomplish_goal` operator. The conditionals are assumed to be

mutually exclusive and exhaustive, and tested in parallel, so their order is supposed to be irrelevant. However, computationally they are evaluated in the order listed, and evaluation will stop at the first one whose condition is true. Due to the parallel assumption, the required execution time is that for a single step, corresponding to the rule whose condition is satisfied. It is a run-time error if no selection rule is satisfied. After the corresponding specific goal is accomplished, then the general goal is reported as 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 and 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` operator 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 you have identified more than one method to accomplish a goal, use that goal as a general goal, and define separate methods to accomplish the more specific goals; use a selection rule set to dispatch control to the specific method. But if you simply want to control which operators in what sequence are executed within a method, use a `Decide`.

2.6 Supplying Auxiliary Information

In order to execute successfully, the methods in a GOMS model often require additional information; this information is auxiliary to the step-by-step procedural knowledge represented directly in the GOMS methods and selection rules, but is logically required for actual tasks to be executed. For example, the benchmark tasks themselves have to be specified, as in the above file-manipulation example. In addition, in order to point to the proper screen object, a method might need to know what color the objects are. To properly compute the execution time required for a command-line file system interface, the exact filenames involved in benchmark tasks need to be specified. If the user is supposed to type in an exact string from memory, this string must be specified somehow.

In many cases, a device simulation will supply the auxiliary information in the form of properties for Visual and Auditory objects that might also specify the tasks for the methods to execute and the specific parameters of the task. For example, the simulated human might be monitoring a simulated radar screen and properly handling whatever targets appear. But constructing an explicit device simulation can be time-consuming, and in the current version of GLEAN, a device simulation must be coded in C++ and compiled and linked together with GLEAN. Furthermore, some auxiliary information, such as knowledge in Long-Term Memory, is not the responsibility of the device, and so must be specified in other ways.

Consequently, GOMSL includes facilities to describe auxiliary information along with the GOMS model. It is thus possible to construct and execute GOMS models with GLEAN without supplying a device simulation. The disadvantage of this approach that the auxiliary information is static; it cannot be changed or updated in response to the simulated user's activity.

The syntax for specifying auxiliary information is based on describing object-like entities with properties and values; these descriptions can appear in the same input file as methods and selection rule sets, and are top-level GOMSL constructs along with methods and selection rule sets. They must not be placed inside methods and selection rule sets, but can appear in any order with them and each other.

Visual Object Description

Visual objects are described outside of any methods as follows:

```
Visual_object: object_name
  property_name is value.
  ...
```

For example, a red button labeled "Start" would be described as:

```
Visual_object: start_button
  Type is Button.
  Label is Start.
  Color is Red.
```

A step like the following will result in `start_button` being stored in WM under the tag `<button>`:

```
Step. Look_for_visual_object_whose Type is Button, and Label is Start
        and_store_under <button>.
```

Subsequently, the following step will point to the button if its color is red:

```
Step. Decide: If Color of <button> is Red,
        Then Point_to <button>.
```

Note that the names for visual objects are chosen by the analyst. GLEAN reserves two property names, `Location` and `Size`, for use by the Visual and Manual processors. At this time, only a device simulation can supply the `Location` and `Size properties` for an object. All other property names and values can be chosen by the analyst.

Auditory Encodings

Auditory encodings simplify the GOMS models for handling auditory input. Speech input items are looked up in the list of encodings, and the corresponding additional properties are added to the representation of the input. The syntax of these encodings is as follows:

```
Auditory_encoding: input_word
  property_name is property_value.
  ...
...
```

For example, suppose a simple form needs to be filled out from speech input in which the name of a field will be given followed by the data to put into the field. The methods for doing the task are greatly simplified if certain words can be recognized as being names of fields. The following supplies a set of encodings that specify this recognition mapping:

```
Auditory_encoding: "Name"
  Type is "field_name".
Auditory_encoding: "Occupation"
  Type is "field_name".
Auditory_encoding: "Income"
  Type is "field_name".
Auditory_encoding: "Default"
  Type is "data".
```

When the auditory processor receives an input utterance, it creates a unique object to represent that input event, and attaches to it the property `Content` with the value being the symbol for the input utterance itself. Thus if the auditory input is the word "Occupation", and object named e.g. "Audobj23" is created whose `Content` property is "Occupation." The processor then searches the set of auditory encodings for a match to the input word and adds to the object whatever property-value pairs are associated with it. In this example, `field_name` would be stored under the property `Type` for object "Audobj23". If no matching encoding is found, then the properties associated with the `Default` encoding are assigned to the input object. Thus, the input words `Name`, `Occupation`, and `Income` are all recognized as having the `Type` of `field_name`. This allows the analyst to write a simple method for responding to any input that names a field_name, instead of checking for each possible content with multiple Decide operators. For example, the methods might include:

```
Step 5. Wait_for_auditory_object_whose Type is "field_name"
        and_store_under <field_name_word>.
Step 6. Look_for_object_whose Label is Content of <field_name_word>
        and_store_under <field_to_fill_in>.
```

Note that the property name `Content` and the encoding object name `Default` are built into GLEAN and must be used for this purpose.

Long-Term Memory Contents

The contents of Long-Term Memory can be specified as a set of concepts (objects) with properties and values. Note the since the value of a property can be the name of another object, complicated information structures are possible. The syntax:

```
LTM_item: LTM_concept
  property_name is property_value.
  ...
...
```

For example, information about the "Cut" command in a simple text editor could be specified as:

```
LTM_item: Cut_Command
  Name is CUT.
  Containing_Menu is Edit.
  Menu_Item_Label is Cut.
  Accelerator_Key is Command-X.
```

Initial Working Memory Contents

When execution of a GOMS model starts, working memory is initially empty unless its initial contents have been specified with an `Initial_WM_contents` description whose syntax is as follows:

```
Initial_WM_contents:
  <tag> is value.
  ...
```

Task Instances

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. A *task instance* is a description of a specific task. It consists of specific values for all of the "parameters" in a task description. A set of task instances can be specified as task item objects whose property values can refer to other objects to form a linked-list sort of structure. The syntax is similar to the above:

```
Task_item: task_item_name
  property_name is property_value.
  ...
...
```

As an example, the file management tasks example above specified a set of four task instances to be executed in a specified sequence. The task items names, `T1`, `T2`, and so forth are arbitrary abbreviations chosen by the analyst. Likewise, the property names and values, such as `Next is None` are chosen by the analyst none of them are fixed by GOMSL or GLEAN.

2.7 Complex Flow of Control

Interrupt Rules

The GOMS model can contain interrupt rules, specified before the first method, as follows:

```
Interrupt_rules
If interrupt_condition Then interupt_operator.
...
```

As many interrupt rules can be listed as desired. The interrupt rules are tested in the order listed, and the action is executed for first one whose condition is true, and the remaining rules are ignored. This test will happen on every step cycle before any regular method steps are executed. If rules actions are executed, such as the execution of a method, then the interrupt processing takes precedence over all other method execution, which is suspended until the interrupt actions are completed. See the section on multithreaded execution for more details. An interrupt_condition is either an ordinary predicate, or the existence predicate that implements a test for the presence of certain objects in the object store:

`Exists <object_tag> predicates`

The name of the first object of which the *`predicates`* are true is stored in the `<object_tag>`, and the `Exists` predicate evaluates to true. If there is no such object, the `Exist` predicate evaluates to false. Normally at least some of the predicates test for properties of the object. For reasons of computational costs, the only objects whose properties will be considered by the `Exist` predicate are those for which an interrupt is psychologically plausible as a capture of attention, in particular, those which are new or color-changed auditory or visual objects. These objects are those that have the `Event_type` property of `New` or `Color_changed`; this property is added to objects by the visual and auditory processors and removed a short time later. No such restriction is placed on the evaluation of ordinary predicates that test only the values of tags.

An *`interupt_operator`* is either an ordinary `Accomplish_goal` operator that begins execution of a submethod, or the `Abort_and_restart` operator (see below).

The following are examples of some interrupt rules from actual models. The first two rules use the `Exist` predicate to test for the existence of an object with the `Event_type` property. The first fires if there is a `New` object with the `Blip` type, and certain other tags have particular values. The second, similar, rule fires if a `Blip` object changes color. The third rule tests simply for a particular combination of tag values and the property of an object specified by tag values.

```
If Exists <new_track> Type of <new_track> is Blip, Event_type of <new_track> is N
<search_window> is OPEN, and <new_track> is_not <last_new_track>,
  Then Accomplish_goal: Notice Radar_Det.

If Exists <new_track> Type of <new_track> is Blip, Event_type of <new_track> is
Color_changed, <search_window> is OPEN, and <new_track> is_not <last_new_track>,
  Then Accomplish_goal: Notice Radar_Det.

If <current_track> is_not Nil, <current_track> is_not Absent, and Status of
<current_track> is Disappearing, and <current_track> is_not <last_disappearing_tr
  Then Accomplish_goal:  Prepare TRK_disappear_cleanup.
```

For compatibility with the multiple-thread system (see below), the interrupt rule set is represented in terms of a special hidden method for the goal of checking interrupts, and which contains a single step consisting of the interrupt rule conditionals themselves. Any methods that might be invoked by interrupt rules are simply ordinary methods. However, since these methods are run at interrupt priority where they lock out normal model processing, normally interrupt methods should be a short and fast-executing as possible.

Multiple Tread Execution

GLEAN allows for multiple execution threads, defined in terms of method execution. That is, execution of a thread begins with the method to accomplish some goal, continues through any submethods that are invoked, and terminates when the initial goal has been accomplished, as indicated by the final `Return` operator. A thread can spawn another thread with the `Also_accomplish_goal` operator; insofar as possible, the steps for all threads are executed in parallel. Interrupt rules and interrupt methods are executed in their own special thread, and take precedence over other threads, as explained below.

> `Also_accomplish_goal:` *`goal`* `as` *`thread_name`*
> `Also_accomplish_goal:` *`goal`* `as` *`thread_name`* `using` *`pseudoargument_tag_list`*
> This operator results in the creation of a new execution thread, whose name is the specified name, whose steps begin to execute in parallel with the other threads present.

During GLEAN execution, at least one thread, the Main thread, exists: this thread is created automatically to accomplish the initial top-level goal, and continues execution until the top-level method for this goal returns with the goal accomplished. If the Main thread spawns any subthreads, then execution will continue until the Main thread and any subthreads have terminated.

If interrupt rules were specified as part of the GOMS model, then a second thread exists during execution, the Interrupt thread. At the beginning of each step cycle, first the Interrupt thread is executed; normally this results only in the testing of the conditions of the interrupt rules. If none are satisfied, then the Main thread and subthread steps are executed. If an interrupt rule is satisfied, then execution of its actions (and any methods it invokes) begins and takes precedence over all other threads - the Main and any of its subthreads are suspended until the action of the interrupt rule is completed, or the goal asserted by the action is accomplished. Then execution of the Main and subthreads resumes.

Finally, if an error exception is raised (see below), an Error thread is created, and begins execution at the next step cycle, and no other threads (not even the interrupt thread) are executed. When the error thread terminates, then on the next step cycle, interrupt and normal execution resume.

A simple thread priority system applies. The Error thread has pre-emptive priority over all other threads, followed by the Interrupt thread, which has priority over all other *normal* threads and will preempt them while executing. The normal threads, including the Main thread, have priority in the order they were created. Since all normal thread steps are executed in parallel, the only relevance of thread priority is in resolving conflicts over access to the processors in the GLEAN architecture. For example, two threads are not allowed to control the Manual motor processor at the same time, so the first thread to execute a manual operator locks out other threads that need to use the manual processor.

More specifically, each step contains information on which processors its operators require. Interstep mental operators such as `Recall_LTM_item` require use of the cognitive processor, but intrastep mental operators do not. On each step cycle, the threads are examined in priority order and the processor requirements of the next step are tested. If the step requires a processor that is currently in use, then that thread's next step is not executed on this cycle. If the required processors are free, then the step is executed, and the operators mark the processors as busy until they are complete. Then the next thread in order is examined. On the next step cycle, the process is repeated. Thus, the highest-priority thread gets first access to any free processors required by its next step, and other threads wait until the processors required by their next step are free. But all threads whose next-step processor requirements are currently met will be executed in simulated parallel.

Although the Error thread and the Interrupt thread lock out all other threads, execution of one of their steps may have to wait until an operator in a normal thread finishes using a processor required by the Error or Interrupt thread. Furthermore, exceptions or interrupts do not interrupt individual operator executions, and errors or interrupting events occurring during a step cycle cannot preempt processing until the current step cycle is complete and a new cycle begins. Thus, each step that starts executing is guaranteed that it and its operators will complete before any other thread is given control of the processors involved.

Error Handling and Exceptions

Based on proposals by Wood (2000), GLEAN includes mechanisms to allow GOMS models to describe how errors would be handled using an approach similar to exceptions in standard programming languages, but with options to allow restarting execution in various ways.

First, each method can specify an error-handling goal. When an error exception is raised, the current thread is suspended, an Error thread is created, and execution begins with the goal specified as the error-handling goal of the most recently started method. The methods invoked by the error-handling goal can use any of the normal operators, and even the Abort_and_restart operator, and in addition, can apply other operators to modify the state of the original thread, and then cause it to resume execution at a different point. Thus, for example, an error-handling method can invoke an Undo function, verify that that the device has been reset to the previous state, and then arrange for the failed method to begin executing again. So that the error-handling method can assess the state of the failed method, when an error exception is raised, several useful tags are deposited in the tag store containing the goal action and object of the failed method, the step in progress when the method failed, and other useful information.

`On_error:` *`error_handling_goal`*

This specification is optional; if present, it must follow the Method_for_goal statement in a method, appearing prior to the first step. For example:

```
Method_for_goal: Select menu_item using <menu_name>, and <menu_item>
  On_error: Retry current_method
  Step 1. Look_for_object_whose Item_name is <menu_name> and_store_under <menu>.
  Step 2. Point_to <menu>.
  ...
```

`Raise` *`exception_name`*

This operator raises an error exception; an error exception can also be raised by one of the normal operators (e.g. a mistyped Keystroke), but this capability is experimental at this time and has not be built into GLEAN. Execution of the method is halted, and the following tags are added to tag store:

`<Exception_name>` - the name specified in the Raise operator.
`<Exception_current_goal_action>` - the action (first) part of the goal at the time of the exception.
`<Exception_current_goal_object>` - the object (second) part of the goal at the time of the exception.
`<Exception_current_step>` - the name of the step being executed when the exception was raised.

If a non-Error thread is executing when an Error exception is raised, normal execution is suspended, and the Error thread is created and started with the On_error: goal specified by the most recently entered method. In addition, the following tag is added to the tag store:

`<Exception_current_thread>` - the name of the thread executing at the time of the exception.

If the Error thread was executing when an Error exception is raised, it means that the attempt to handle an error has failed, and more drastic recovery must be attempted. The original failed thread is unwound until a higher level On_error goal is found; that is, starting at the bottom of the goal and method hierarchy, methods are terminated and goals are discarded until we reach the next level up that has a new error-recovery goal. The Exception tags are updated to reflect the goal and step of this higher-level method, and the name of the new exception. The Error thread is completely unwound, and execution restarted with the new error goal.

When the Error thread terminates (the error-handling goal is accomplished), the Exception information tags deposited by the Raise operator are automatically deleted from the tag store.

Error handling methods can use the following operators to modify how execution of the original failed thread will be resumed.

`Abort_and_restart`

As in the case of the interrupt rules, all currently executing methods and threads are terminated, and execution begins again with the top-level method for the initial goal. Except for the Exception information tags, the

contents of the tag store and the object store are left in their current state.

`Stop_with_message` *`message`*
The specified message (an identifier) is output and the simulation is stopped.

`Resume` *`thread`* `accomplishing_goal` *`goal_action goal_object`* `at` *`stepname`*
The named thread is unwound to the specified goal, and execution resumes at the designated step with any intervening steps skipped over. It is assumed that the tag and object stores contain whatever information is necessary to successfully resume the specified method at the specified step. The thread name, goal action and object, and step name can be taken either from the Exception information tags stored by the Raise operator, or can be specified by "hard coding" or some other means.

2.8 High-Level GOMS Models

Once a set of functions have been chosen, a candidate interface design can be developed, and GOMS model can be used to determine whether a user will be able to accomplish a specified set of goals using the system and its interface, and predictions of learning difficulty and performance can be made using the approach described in this document, assisted by the GLEAN tool. However, if the system functionality has been poorly chosen, the resulting system is unlikely to be adequately useful or usable no matter how much effort is put into designing the interface - if the underlying functionality will not support the user properly, making that functionality easy and fast to access simply won't help. Kieras (2004) provides several examples of such failures of functionality design; many more could be listed. Sometimes the functionality failures are due to limitations in the computer science technology, either economic or technical; products can and have been proposed that simply are not possible. However, often, as in the examples cited by Kieras (2004), the failure is due to inadequate analysis of what is required to support users, leading to products that were essentially useless in spite of considerable development effort and a high price.

Kieras (2004) proposed *High-Level GOMS Models* as an approach to the *design of functionality*, the problem of how the functions of a system should be chosen to maximize both the usefulness and usability of the system. High-level GOMS models are a technique that can be used to aid in the design of functionality. A high-level GOMS model is basically a GOMS model that does not contain any methods that are specific to a particular interface; rather the human is assumed to interact with the device through *high-level operators* that specify only the information exchanged between the human and device, without any specifics of how the communication will take place. The simulated human can *put* to the device a function invocation specification that causes the device to perform some function, basically changing the device's state. The function invocation takes the form of a list of properties and values that are interpreted by the device according to a candidate functionality design. The simulated human can *get* from the device a set of specified information; this specification is also in the form of a list of properties and values; the device responds by sending back an information object with a set of properties and values that can then be interrogated by the simulated human. Thus the communication between the human and the device is via an abstract information pathway, the *high-level processor*, that does not correspond to any actual human sensory-motor modalities. Roughly speaking, this processor represents a direct connection between the internal functionality of the device and the user's cognitive processor; of course, no such actual "telepathic" connection exists, but this is still a use abstraction for analysis - if there is no interface problem, how best could the user and device interact? In this way, the basic design concept for the system and how it can be used to accomplish task goals, can be explored prior to any commitments about how the system interface will actually works. So that the execution time of these operators will be on the same scale as actual interface methods, these operators take a relatively long time (1 sec) to execute, but these times are purely placeholders.

A poor choice of device functionality will show up as slow, clumsy, or complex high-level user methods, while a good choice will result in simple and straightforward high-level methods. As the design is elaborated, the high-level operators will be replaced with the specific interaction methods for controlling the device and access information from its display.

The following operators are available to support communicating with the device in terms of high-level operators:

`Put_to_device property is value... .`

The device is sent the list of property-value pairs after 1 sec. GOMSL and GLEAN have no restrictions on what the property-value pairs are. However, a useful convention is that the first property is `Function` whose value is the name of a device function; the additional property value-pairs are then parameters for the function. The effects of this function invocation are completely determined by the simulated device code. The behavior of the default dummy device is to simply print out the property-value pairs.

`Get_from_device property is value, ... and_store_under <tag>.`

The device is sent the list of property-value pairs after 1 sec. GOMSL and GLEAN have no restrictions on what the property-value pairs are. However, a useful convention is that the first property is `Data` whose value is the name of the kind of data desired from the device, and any additional parameters specify additional attributes of the desired data. The device will respond by sending to GLEAN's *High-Level Processor* an object whose name is stored in the `<tag>` and which has whatever property-value pairs specified by the device. The property values stored on the tagged object can then be examined in the methods to make use of the information supplied by the device.

Simple Example of High-Level GOMS Models

This example is based on an example in Kieras (2004) concerning the lack of functionality in the first generation of PDAs, originally called "digital diaries." These devices were essentially little more than an electronic version of a paper calendar, and lacked critical functions that we take for granted today. The PDA device functionality is drastically simplified for purposes of this example. The device keeps a calendar consisting of meetings scheduled on days of a month with a designated topic. So a meeting is just a `<day, topic>` pair. The user can enter a meeting into the device by specifying a day and a meeting topic. Whenever a meeting scheduling function is invoked, the demo device prints out its current calendar just to allow us to confirm the state of the device's internal data structure for demonstration purposes. The simulated user does not “see” this display - no actual interface display is being assumed.

Two levels of device functionality are compared in this example. The Low-Functionality (LF) device has a `Schedule_meeting` function that takes parameters for the day and the meeting topic. The user can invoke this function with the two parameters, and the device will update its internal data structure. The LF device can also identify the day number of the next day, or of a week later - that is, the user can request that the device present one of these future dates from a supplied date. For example, an actual interface could do this just by displaying the conventional calendar display, where one looks to the next cell to the right to get the date of the next day, or down one row to get the date a week later. This function, which paper calendars supply, makes it easier for the user to figure out the date for the next meeting.

The methods for the High-Level GOMS model for using the LF version of the device is shown below. One particular scheduling task is shown as the initial Working Memory contents. The method shows that to schedule this repeating meeting using the LF device, the user must invoke the schedule function for each meeting, then ask the device for the date of the next meeting, schedule it, and then repeat. With an appropriate simulated device that responds to the High-Level operators, GLEAN shows that the simulated user would take about 13.5 seconds to complete the task, using the place-holder execution time values for the High-Level Operators. Clearly, this time would be longer the more meetings that need to be scheduled, and in an actual interface might be considerably longer, depending on the details of the interface and its procedures.

```
Define_model: "HL Methods for low functionality calendar device"
Starting_goal is Schedule Repeating_Meeting.

Initial_WM_contents:
  <starting_date> is "3".
  <ending_date> is "24".
  <topic> is "Project Review".
```

```
  <repeat_type> is Weekly.

Method_for_goal: Schedule Repeating_Meeting
Step 1. Store <starting_date> under <date>.
Step 2. Decide: If <date> is_greater_than <ending_date>,
  Then Return_with_goal_accomplished.
Step 3. Put_to_device Function is Schedule_meeting, Date is <date>, Topic is <top
Step 4. Decide:
  If <repeat_type> is Daily, Then Get_from_device Data is Next_date,
          Current_date is <date>, Increment is One_day and_store_under <result>;
  If <repeat_type> is Weekly, Then Get_from_device Data is Next_date,
          Current_date is <date>, Increment is One_week and_store_under <result>
Step 5. Store Next_date of <result> under <date>.
Step 6. Goto 2.
```

The High-Functionality (HF) device has a `Schedule_repeating_meeting` function. The user supplies the startin date, ending date, topic, and meeting increment (`Daily` or `Weekly`); the device computes the date of each meeting automatically and adds it to the calendar. Below is shown the corresponding High-Level GOMS model methods for using the HF version of the device. Since all the simulated user has to do is invoke a single function, according to the GLEAN execution results the task would take only 1.1 sec, and will be independent of the number of meetings needing to be scheduled.

```
Define_model: "HL Methods for high functionality calendar device"
Starting_goal is Schedule Repeating_Meeting.

Initial_WM_contents:
  <starting_date> is "3".
  <ending_date> is "24".
  <topic> is "Project Review".
  <repeat_type> is Weekly.

Method_for_goal: Schedule Repeating_Meeting using <starting_date>, <ending_date>,
  <topic>, and <repeat_type>
Step 1. Put_to_device Function is Schedule_repeating_meeting, Date is <starting_c
  Topic is <topic>, Repeat_specification is <repeat_type>, and  Ending_date is
<ending_date>.
Step 2. Return_with_goal_accomplished.
```

Notice how the comparison between the methods for the two levels of functionality can be performed both qualitatively, by comparing the content and structure of the GOMSL methods, and quantitatively, by comparing the very approximate, but well-defined, execution time for the methods. This comparison does not involve any assumptions about the specific interface or interface methods involved in the device. Thus the High-Level GOMS model represents the basic structure of how the user interacts with the device functionality. By iterating over combinations of proposed functionality and High-Level user methods, the analyst can consider different allocations of function between the user and the system, and choose the best set of system functions and the user's top-level methods prior to any specific interface design. This enables a rigorous and quantitative form of Kieras's (2004) proposal for an approach to the design of functionality.

3. GENERAL ISSUES IN GOMS MODELING

In constructing a GOMS model, the analyst is engaging in a particular form of task analysis, in which the informal intuitions or empirical data about how people perform (or will perform) a task is represented in a relatively rigorous form that has some useful predictive features (see Kieras, 2004, for more discussion). To provide a more complete presentation of GOMS modeling, this section addresses some general issues concerning the task-analysis component of constructing a GOMS model which are shared with other forms of task analysis, especially Hierarchical Task Analysis (see Kirwan & Ainsworth, 1992; Annett, Duncan, Stammers, & Gray, 1971).

3.1 Judgment Calls

In constructing a GOMS model, the analyst is relying on a task analysis that involves judgments 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.

These are standard problems in task analysis (see Kieras, 2004, Kirwan & Ainsworth, 1992; Annett, et al., 1971). It is possible to collect extensive behavioral data on how users view and decompose tasks, but often it is not practical to do so because of time and cost constraints on the interface design process. Instead, the analyst must often 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 speculating on a psychological theory or model for how people do the task, and so will have to make 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 did not include separate cut and paste operations. 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 the subgoals 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, if the analyst's assumptions are based on a careful consideration from the user's point of view, they 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. The user, even a very experienced one, is not necessarily a source of "truth" about the system or the tasks (cf. Annett, et al., 1971). 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 suboptimal 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. One approach is to bypass the analysis of a complex process by simply representing it with a "dummy" or "placeholder" operator, such as the `Think_of` operator in GOMSL. In this way the analyst documents the presence of the process, and can consider what influence it might have on the user's performance with a design. A more flexible approach is the "yellow pad" heuristic. Suppose the user has already done the complex processing and has written the results down on a yellow note pad and simply refers to them along with the rest of the information about the task instance.

For example, in MacWrite, the user may use tabs to control the layout of a table. 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 instance description supplied to the methods. (cf. the discussion in Bennett, Lorch, Kieras, & Polson, 1987). The analyst uses the GOMSL `Get_task_item` operator to represent when this information is accessed.

As a second example, consider a word-processor user who is making changes in a document from a marked-up hard-copy. 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 the `Get_task_item` operator to access the task information. 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, 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 it is to think up good find-strings. In this case, bypassing a process might produce seriously misleading results.

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 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 the specific Keystroke-Level operations appearing in the trace. 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 designated `<filename> by` followed by <CR>. This corresponds to a user who knows how to use the system in the general way normally intended. Such GOMS models are *generative* - rather than being limited to specific snippets of behavior, they can generate all possible traces from a single set of methods. This is a critical advantage of GOMS models and other cognitive-architecture models (see John & Kieras, 1996b; Kieras, Wood, & Meyer, 1997; for more discussion).

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). 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. You can then use the GLEAN tool to compute execution time predictions for as many different benchmark tasks as desired.

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. See Kieras(2004) for more discussion of this approach. Clearly, the designer and analyst must closely cooperate, or be 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

A GOMS analysis of a task follows the familiar top-down decomposition approach. The model is developed top-down from the most general user goal to more specific subgoals, with primitive operators finally at the bottom. The methods for the goals at each level are dealt with before going down to a lower level. The recipe presented here thus follows 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".
... 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"`) 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. In a future version of GLEAN, these analyst-defined placeholders can be assigned an assumed execution time. We would then go on to describe in detail the methods involved in dealing with the mail program.

4.1 Summary of Procedure

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 Document
  Step.Store First under <current_task_name>.
  Step Check_for_done.
  Decide: If <current_task_name> is None, Then
          Delete <current_task>; Delete <current_task_name>;
          Return_with_goal_accomplished.
  Step. Get_task_item_whose Name is <current_task_name>
          and_store_under <current_task>.
  Step. Accomplish_goal: Perform Unit_task.
  Step. Store Next of <current_task> under <current_task_name>;
          Goto Check_for_done.
```

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_rules_for_goal: Perform Unit_task
  If Type of <current_task> is move,
    Then Accomplish_goal: Move Text.
  If Type of <current_task> is delete,
    Then Accomplish_goal: Erase Text.
  If Type of <current_task> is copy,
    Then Accomplish_goal: Copy Text.
```

```
//... 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 GOMS model 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."
- Define new high-level operators, and bypass complex psychological processes as needed. Make a note of the analyst-defined 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. Keystroke CR.
  Step 4. Verify "main menu is present".
  Step 5. Keystroke F1.
  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). GOMSL imposes some guidelines; for example, only one operator like Keystroke or Point_to is allowed in a 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 `Look_for` operator executed prior to a `Point_to`, 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 where 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 `Get_task_item` operator that begins each task.

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 what text will be selected, where it will be moved to, 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 analyst defined operators, showing which methods each operator appears in. Check for consistency of terminology and usage. Redefine new or old operators to ensure consistency; and add new operators to the list. For example, instead of `Verify "results are correct"` we could end up with:

  ```
  Verify "results are right".
  Think_of "whether desired results have been obtained".
  Verify "OK".
  ```

- 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 selected text is:

```
Method_for_goal: Copy Selection
Step 1. Select Text.
Step 2. Issue Command using Copy.
Step 3. Return_with_goal_accomplished.
```

We choose to provide a method for the Step 1 operator `Select Text.` We rewrite the method as:

```
Method_for_goal: Copy Selection
Step 1. Accomplish_goal: Select Text.
Step 2. Issue Command using Copy.
Step 3. Return_with_goal_accomplished.
```

Then we provide a method for the goal of selecting the text. To make the example clear, the changed line is shown in **Boldface**.

Step C. Documenting and Checking the Analysis

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

- Analyst-defined operators used, along with a brief description of each one
- Assumptions and judgment calls that you made during the analysis

Then, choose some representative task instances, and check on the accuracy of the model either by hand or with the GLEAN tool, to verify that the methods generate the correct sequence of overt actions. 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 An Example of Using the Procedure

This example shows the use of GOMSL 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**. There are several simplifications in this example, such as assuming that no scrolling is necessary, and locations in text are represented in an highly simplified fashion.

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. We are assumed a task representation which we make up as we go along; the final task representation will be presented later.

```
Method_for_goal: Edit Document
  Step.Store First under <current_task_name>.
  Step Check_for_done.
  Decide: If <current_task_name> is None, Then
           Delete <current_task>; Delete <current_task_name>;
           Return_with_goal_accomplished.
  Step. Get_task_item_whose Name is <current_task_name>
           and_store_under <current_task>.
  Step. Accomplish_goal: Perform Unit_task.
  Step. Store Next of <current_task> under <current_task_name>;
           Goto Check_for_done.

Selection_rules_for_goal: Perform Unit_task
  If Type of <current_task> is move,
    Then Accomplish_goal: Move Text.
  If Type of <current_task> is delete,
    Then Accomplish_goal: Erase Text.
  If Type of <current_task> is copy,
    Then Accomplish_goal: Copy Text.
  //... etc. ...
  Return_with_goal_accomplished.
```

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 Document
  Step.Store First under <current_task_name>.
  Step Check_for_done.
  Decide: If <current_task_name> is None, Then
           Delete <current_task>; Delete <current_task_name>;
           Return_with_goal_accomplished.
  Step. Get_task_item_whose Name is <current_task_name>
           and_store_under <current_task>.
  Step. Accomplish_goal: Perform Unit_task.
  Step. Store Next of <current_task> under <current_task_name>;
           Goto Check_for_done.

Selection_rules_for_goal: Perform Unit_task
  If Type of <current_task> is move,
    Then Accomplish_goal: Move Text.
  If Type of <current_task> is delete,
    Then Accomplish_goal: Erase Text.
  If Type of <current_task> is copy,
```

```
    Then Accomplish_goal: Copy Text.
  //... etc. ...
  Return_with_goal_accomplished.
```

```
Method_for_goal: Move Text
  Step 1. Cut Selection.
  Step 2. Paste Selection.
  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 selected text. 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 a selection is a sketch that leaves out the details and is not in formal GOMSL. It is also too long, as explained in the guidelines. The problem is fixed in the second draft, and the details are filled in and expressed in formal GOMSL To save space, the top-level method and selection rule will not be shown again until the final version of the model.

```
Method_for_goal: Move Text
  Step 1. Accomplish_goal: Cut Selection.
  Step 2. Accomplish_goal: Paste Selection.
  Step 3. Verify "correct text moved".
  Step 4. Return_with_goal_accomplished.
```

```
Method_for_goal: Cut Selection // First draft
  Step 1. Look_for start of selection.
  Step 2. Point_to it.
  Step 3. Hold_down mouse_button.
  Step 4. Look_for end of selection.
  Step 5. Point_to it.
  Step 6. Release mouse_button.
  Step 7. Verify "correct text is selected".
  Step 8. Point_to Edit menu bar item.
  Step 9. Hold_down mouse_button.
  Step 10. Point_to CUT item
  Step 11. Release cursor button.
  Step 12. Return with goal accomplished
```

This method has too many steps, according to the guidelines. Also, this is only the second level of goals, and the method already has external primitive operators. Notice that Steps 1-7 correspond to a general method for how many things are selected almost everywhere on the Macintosh, and Steps 8-11 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 Selection // Second draft
  Step 1. Select Text.
```

```
  Step 2. Issue Cut_Command.
  Step 3. Return_with_goal_accomplished.

Method_for_goal: Paste Selection
  Step 1. Select Insertion_point.
  Step 2. Issue Paste_Command.
  Step 3. Return_with_goal_accomplished.
```

Pass 4

We now provide several methods at once, those 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. Since we can see that there is going to be some commonality in the methods for issues these two commands, we assume that they can be replaced with a single Issue Command method that is given the "name" or "concept" of the desired command and makes the proper menu accesses. The method is called with a pseudoargument, and will be defined later with the corresponding pseudoparameter. We make the simplifying assumption that our user does not make use of the command-key shortcuts.

```
Method_for_goal: Move Text
  Step 1. Accomplish_goal: Cut Selection.
  Step 2. Accomplish_goal: Paste Selection.
  Step 3. Verify "correct text moved".
  Step 4. Return_with_goal_accomplished.

Method_for_goal: Cut Selection
  Step 1. Accomplish_goal: Select Text.
  Step 2. Accomplish_goal: Issue Command using Cut.
  Step 3. Return_with_goal_accomplished.

Method_for_goal: Paste Selection
  Step 1. Accomplish_goal: Select Insertion_point.
  Step 2. Accomplish_goal: Issue Command using Paste.
  Step 3. Return_with_goal_accomplished.

// Each task specifies the "size" of the text involved
Selection_rules_for_goal: Select Text
  If Text_size of <current_task> is Word,
    Then Accomplish_goal: Select Word.
  If Text_size of <current_task> is Arbitrary,
    Then Accomplish_goal: Select Arbitrary_text.
  Return_with_goal_accomplished.

// The task specifies the to-be-selected word
Method_for_goal: Select Word
  Step 1. Look_for_object_whose Content is Text_selection of <current_task>
           and_store_under <target>.
  Step 2. Point_to <target>; Delete <target>.
  Step 3. Double_click mouse_button.
  Step 4. Verify "correct text is selected".
  Step 5. Return_with_goal_accomplished.

// The task specifies the beginning and ending word of the text
Method_for_goal: Select Arbitrary_text
  Step 1. Look_for_object_whose
           Content is Text_selection_start of <current_task>
           and_store_under <target>.
```

```
    Step 2. Point_to <target>.
    Step 3. Hold_down mouse_button.
    Step 4. Look_for_object_whose Content is
             Text_selection_end of <current_task>
             and_store_under <target>.
    Step 5. Point_to <target>; Delete <target>.
    Step 6. Release mouse_button.
    Step 7. Verify "correct text is selected".
    Step 8. Return_with_goal_accomplished.

Method_for_goal: Select Insertion_point
    Step 1. Look_for_object_whose
             Content is Text_insertion_point of <current_task>
             and_store_under <target>.
    Step 2. Point_to <target>; Delete <target>.
    Step 3. Click mouse_button.
    Step 4. Verify "insertion cursor is at correct place".
    Step 5. Return_with_goal_accomplished.
```

The method for selecting arbitrary text seems somewhat long. This is the result of a judgment call that the user has find on the screen the word specified as the beginning of the to-be-selected text (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_over` 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 method is the generic command-issuing method. In order to pick the proper menu command, the user must first remember which menu to open, find it on the screen, open it, and then find and select the actual menu item:

```
// Assumes that user does not use command-key shortcuts
Method_for_goal: Issue Command using <command_name>
// Recall which menu the command is on, find it, and open it
    Step 1. Recall_LTM_item_whose
             Name is <command_name>
             and_store_under <command>.
    Step 2. Look_for_object_whose
             Label is Containing_Menu of <command>
             and_store_under <target>.
    Step 3. Point_to <target>.
    Step 4. Hold_down mouse_button.
    Step 5. Verify "correct menu appears".
// Now select the menu item for the command
    Step 6. Look_for_object_whose
             Label is Menu_Item_Label of <command>
             and_store_under <target>.
    Step 7. Point_to <target>.
    Step 8. Verify "correct menu command is highlighted".
    Step 9. Release mouse_button.
    Step 10.Delete <command>; Delete <target>;
             Return_with_goal_accomplished.
```

Finishing the Model

For completeness in this example, the methods for deleting and duplicating text have been added. Often, writing the methods for additional goals is quite easy once the first set of methods have been written – the lower-level

submethods are simply reused in different combinations or with different commands. We then examine the methods, and locate the task information that the methods require, and reconcile and revise it as necessary. You may find it helpful not to be as precise about the task information as the above example on the first pass.

In addition, the auxiliary information can be collected and specified at this time. For the GLEAN tool to execute these methods in the absence of a MacWrite simulator, there needs to be some visual objects for the methods to look for and point at. These objects need only be minimal or "dummy" objects. Finally, we specify the LTM items required by the Issue Command method.

Final GOMS Model

Below is the final version of the GOMS model. The intermediate versions produced in the separate passes above are discarded. A set of tasks are shown, named T1 through T4, chained together in linked-list style to show the order of execution.

```
Define_model: "MacWrite Example"
  Starting_goal is Edit Document.

Task_item: T1
  Name is First.
  Type is copy.
  Text_size is Word.
  Text_selection is "foobar".
  Text_insertion_point is "*".
  Next is T2.

Task_item: T2
  Name is T2.
  Type is copy.
  Text_size is Arbitrary.
  Text_selection_start is "Now".
  Text_selection_end is "country".
  Next is T3.

Task_item: T3
  Name is T3.
  Type is delete.
  Text_size is Word.
  Text_selection is "foobar".
  Text_insertion_point is "*".
  Next is T4.

Task_item: T4
  Name is T4.
  Type is move.
  Text_size is Arbitrary.
  Text_selection_start is "Now".
  Text_selection_end is "country".
  Next is None.


// Dummy visual objects - targets for Look_for and Point_to
Visual_object: Dummy_text_word
  Content is "foobar".
Visual_object: Dummy_text_selection_start
  Content is "Now".
```

```
Visual_object: Dummy_text_selection_end
  Content is "country".
Visual_object: Dummy_text_insertion_point
  Content is "*".

// Minimal description of the visual objects in the editor interface
Visual_object: Edit_menu
  Label is Edit.
Visual_object: Cut_menu_item
  Label is Cut.
Visual_object: Copy_menu_item
  Label is Copy.
Visual_object: Paste_menu_item
  Label is Paste.

// Long-Term Memory contents about which items are in which menu
LTM_item: Cut_Command
  Name is Cut.
  Containing_Menu is Edit.
  Menu_Item_Label is Cut.
  Accelerator_Key is COMMAND-X.
LTM_item: Copy_Command
  Name is Copy.
  Containing_Menu is Edit.
  Menu_Item_Label is Copy.
  Accelerator_Key is COMMAND-C.
LTM_item: Paste_Command
  Name is Paste.
  Containing_Menu is Edit.
  Menu_Item_Label is Paste.
  Accelerator_Key is COMMAND-V.

// Top-Level Unit Task Method
Method_for_goal: Edit Document
  Step.Store First under <current_task_name>.
  Step Check_for_done.
  Decide: If <current_task_name> is None, Then
          Delete <current_task>; Delete <current_task_name>;
          Return_with_goal_accomplished.
  Step. Get_task_item_whose Name is <current_task_name>
          and_store_under <current_task>.
  Step. Accomplish_goal: Perform Unit_task.
  Step. Store Next of <current_task> under <current_task_name>;
          Goto Check_for_done.

Selection_rules_for_goal: Perform Unit_task
  If Type of <current_task> is move,
    Then Accomplish_goal: Move Text.
  If Type of <current_task> is delete,
    Then Accomplish_goal: Erase Text.
  If Type of <current_task> is copy,
    Then Accomplish_goal: Copy Text.
  //... etc. ...
  Return_with_goal_accomplished.

Method_for_goal: Erase Text
```

```
  Step 1. Accomplish_goal: Select Text.
  Step 2. Keystroke DELETE.
  Step 3. Verify "correct text deleted".
  Step 4. Return_with_goal_accomplished.

Method_for_goal: Move Text
  Step 1. Accomplish_goal: Cut Selection.
  Step 2. Accomplish_goal: Paste Selection.
  Step 3. Verify "correct text moved".
  Step 4. Return_with_goal_accomplished.

Method_for_goal: Copy Text
  Step 1. Accomplish_goal: Copy Selection.
  Step 2. Accomplish_goal: Paste Selection.
  Step 3. Verify "correct text moved".
  Step 4. Return_with_goal_accomplished.

Method_for_goal: Cut Selection
  Step 1. Accomplish_goal: Select Text.
  Step 2. Accomplish_goal: Issue Command using Cut.
  Step 3. Return_with_goal_accomplished.

Method_for_goal: Copy Selection
  Step 1. Accomplish_goal: Select Text.
  Step 2. Accomplish_goal: Issue Command using Copy.
  Step 3. Return_with_goal_accomplished.

Method_for_goal: Paste Selection
  Step 1. Accomplish_goal: Select Insertion_point.
  Step 2. Accomplish_goal: Issue Command using Paste.
  Step 3. Return_with_goal_accomplished.

// Each task specifies the "size" of the text involved
Selection_rules_for_goal: Select Text
  If Text_size of <current_task> is Word,
    Then Accomplish_goal: Select Word.
  If Text_size of <current_task> is Arbitrary,
    Then Accomplish_goal: Select Arbitrary_text.
  Return_with_goal_accomplished.

// The task specifies the to-be-selected word
Method_for_goal: Select Word
  Step 1. Look_for_object_whose Content is Text_selection of <current_task>
          and_store_under <target>.
  Step 2. Point_to <target>; Delete <target>.
  Step 3. Double_click mouse_button.
  Step 4. Verify "correct text is selected".
  Step 5. Return_with_goal_accomplished.

// The task specifies the beginning and ending word of the text
Method_for_goal: Select Arbitrary_text
  Step 1. Look_for_object_whose
          Content is Text_selection_start of <current_task>
          and_store_under <target>.
  Step 2. Point_to <target>.
  Step 3. Hold_down mouse_button.
```

```
  Step 4. Look_for_object_whose Content is
           Text_selection_end of <current_task>
           and_store_under <target>.
  Step 5. Point_to <target>; Delete <target>.
  Step 6. Release mouse_button.
  Step 7. Verify "correct text is selected".
  Step 8. Return_with_goal_accomplished.

Method_for_goal: Select Insertion_point
  Step 1. Look_for_object_whose
           Content is Text_insertion_point of <current_task>
           and_store_under <target>.
  Step 2. Point_to <target>; Delete <target>.
  Step 3. Click mouse_button.
  Step 4. Verify "insertion cursor is at correct place".
  Step 5. Return_with_goal_accomplished.
// Assumes that user does not use command-key shortcuts

Method_for_goal: Issue Command using <command_name>
// Recall which menu the command is on, find it, and open it
  Step 1. Recall_LTM_item_whose
           Name is <command_name>
           and_store_under <command>.
  Step 2. Look_for_object_whose
           Label is Containing_Menu of <command>
           and_store_under <target>.
  Step 3. Point_to <target>.
  Step 4. Hold_down mouse_button.
  Step 5. Verify "correct menu appears".
// Now select the menu item for the command
  Step 6. Look_for_object_whose
           Label is Menu_Item_Label of <command>
           and_store_under <target>.
  Step 7. Point_to <target>.
  Step 8. Verify "correct menu command is highlighted".
  Step 9. Release mouse_button.
  Step 10.Delete <command>; Delete <target>;
           Return_with_goal_accomplished.
```

Step C: Documenting and Checking the Analysis

Running the GLEAN tool on the model revealed syntax errors and bugs in the methods, which were then corrected so that the model generated the proper sequence of inputs to the dummy device. The following are the assumptions and judgment calls associated with this model.

- 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 commands, except for deleting text, where it is assumed that the Delete keystroke is preferred.

- When selecting an arbitrary piece of text, users view as two separate steps locating the beginning point and putting the cursor there. Likewise, they locate the end point and put the cursor there in two separate steps. There are plausible alternative judgment calls.
- The model does not elaborate on the issues involved with choosing the beginning and ending points for text selection, or the point for positioning the insertion cursor. It is assumed that each task supplies the identity of a screen object (such as a word) that can be pointed to, but does not address the fact that actually the cursor would be pointed just before, after, or in-between words.

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 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

GOMSL and GLEAN 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 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 GOMSL learning units (defined below) in the complete GOMS model for the interface – basically, the number of GOMSL statements. 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 GOMSL 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 GOMSL 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, the key variable is the number of GOMSL 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, the key factor is which and how many statements have to be passed through in order to perform the task.

Time Estimates Require 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 (see Section 2.8) 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, although the High-Level operators have been given an arbitrary "middle-size" execution time to allow rigorous, but extremely approximate comparisons between High-Level GOMS models. To help make the point about the cautions needed for making time estimates for models involving only high-level operators, 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.
Step 4. Debug_program.
Step 5. Return_with_goal_accomplished.
```

Suppose we do not decompose these very high-level operators any further, 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 finding an icon on the screen, can be assumed to be already known to the user, and have relatively small, constant, and known execution times. Furthermore, unlike the approach described in Section 2.8 and Kieras(2004), this high-level model does not incorporate any assumed system functionality that would define a finer grain to these steps, which would justify an approximate analysis. In contrast, if the methods have been represented at the standard primitive operator level, the calculations for predicting learning and execution times will produce reasonable accurate and useful results.

Estimating Learning Time

Counting GOMSL learning units. The GLEAN tool predicts the the learning times and transfer of training based on the number of production rules, called learning units here, that underlie the GOMSL representation. The number of learning units are defined as follows:

- A `Step` statement counts as one unit regardless of the number or kind of operators:

```
Step n. operators
```

- The method and selection rule set statement is also be counted as one unit:

```
Method_for_goal: goal
Selection_rule_set_for_goal: general_goal
```

- Each `If-Then` conditional statement used in a selection rule counts as one unit:

```
If condition Then Accomplish_goal: specific_goal.
```

- The terminating statement of a selection rule set counts as one unit:

```
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 the number of `If-Then`s or `Else`s contained in the `Decide` operator is the number of units for the step. Thus, the following steps contain one, two, and three units, respectively:

```
Step 1. Decide: If predicates Then operators.
Step 2. Decide: If predicates Then operators Else operators.
Step 3. Decide:
   If predicates Then operators;
   If predicates Then operators;
   Else operators.
```

The GLEAN tool applies these definitions when it calculates the number of learning units for a GOMS model.

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 task instances 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 tasks to the pure learning time. 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 and GOMSL was defined so that the relation of GOMS model statements to production rules was known (the learning units defined above), and so the pure learning time can be estimated from the GOMSL methods following the learning unit counting rules. The GLEAN tool performs this calculation, showing the number of learning units associated with each method and the total for the GOMS model. This total is used to determine the pure method learning time.

But the time required to learn a single GOMSL unit 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 × Number of GOMSL learning units

Where:

Learning Time Parameter = *30 sec for rigorous procedure training*

17 sec for a typical learning situation

This learning time per unit parameter is inherently elusive, so the absolute values of these learning time estimates must be taken with several grains of salt. Comparing them for e.g. two alternative designs is much more reliable. We can expect further refinement and modifications to this parameter as more research is done.

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 LTM item
- one chunk for each property-value pair defined for an LTM item
- one chunk for each familiar unit in the value in a property-value pair.

For example, suppose the LTM item is from the above MacWrite example:

```
LTM_item: Cut_Command
  Name is Cut.
```

```
Containing_Menu is Edit.
Menu_Item_Label is Cut.
Accelerator_Key is COMMAND-X.
```

There is one chunk for the LTM_item itself, 4 chunks for the four property-value pairs, three chunks for the values in the first three pairs, and two chunks for the `COMMAND-X` value in the last pair. This gives a total of 10 chunks, giving 60 s pure learning time for this collection of LTM information. Add the estimated time to the total learning time.

Do not count LTM storage time if the item is already known. For example, the above LTM item is standard on MacOS, and so would probably be known to an experienced Mac user.

Estimating learning benefits from prior knowledge and consistency. 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 GOMSL units to be learned.

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 that are used everywhere they are appropriate, and few or no special case methods are required. (See the Section 1.2 file manipulation example, and the issue-command method in the MacWrite example.) 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 very 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 GOMSL 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. The GLEAN tool performs this calculation. 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 GOMSL 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 identical operators and arguments. The research on transfer of procedure learning indicates that the transfer process is not very robust; small deviations are enough to cause people not to see the commonality. You may find that your methods are very similar but not quite similar enough due to accidental differences in wording; it is appropriate to rewrite the methods to bring out the similarity, as long as you are confident that you are not misrepresenting the procedures that you believe the users actually know.

The GLEAN tool performs the transfer calculation by analyzing the methods and selection rules in the order they are listed in the GOMS model input file. Each method or selection rule is compared with those that have appeared previously in the model. If the goal of the method is similar to the goal of a previous method, then the steps are checked for similarity along the lines described above. Similar steps are counted; if at least one step is similar, then the method statement is also included. The process is repeated for each previous method; the previous method that produces the greatest similarity is used for the final calculation result. GLEAN presents this result in terms of the number of learning units transferred from the source method, and the number of learning units remaining for the method. Note that the final total number of learning units under this process is invariant with the order in which the methods are analyzed; the input order is used simply to assist the analyst in interpreting the results.

As an example of a learning and transfer calculation, GLEAN produced the following result for the MacWrite example above:

```
:Models:MacWrite_example.gomsl
LEARNING ANALYSIS
------------------------------------------------------------------------------
Method           Units:  number   learned  to learn   Source Method
------------------------------------------------------------------------------
Edit    Document            6         0         6
Perform Unit_task           5         0         5
Move    Text                5         0         5
Copy    Text                5         0         5
Erase   Text                5         0         5
Cut     Selection           4         0         4
Copy    Selection           4         4         0   Cut     Selection
Paste   Selection           4         0         4
Select  Text                4         0         4
Select  Word                6         0         6
Select  Arbitrary_text      9         0         9
Select  Insertion_point     6         0         6
Issue   Command            11         0        11
------------------------------------------------------------------------------
Totals:                    74         4        70
```

This model was written with many shared submethods, so the total number of learning units is only 74, the total in the first column. But the similarity analysis shows that all 4 units of the `Copy Selection` method were already learned from the previously appearing `Cut Selection` method, meaning that this method required zero learning time. The result is that final total of 70 units to learn. Examining the two methods shows why the transfer could occur:

```
Method_for_goal: Cut Selection
  Step 1. Accomplish_goal: Select Text.
  Step 2. Accomplish_goal: Issue Command using Cut.
  Step 3. Return_with_goal_accomplished.

Method_for_goal: Copy Selection
  Step 1. Accomplish_goal: Select Text.
  Step 2. Accomplish_goal: Issue Command using Copy.
  Step 3. Return_with_goal_accomplished.
```

These two methods have goals that differ only in the verb term, `Cut` vs. `Copy`. If these terms are replaced with `X` throughout the two methods, then all four GOMSL statements are identical in the two methods. As a negative example, consider the apparent similarity between the method for cutting a selection and pasting a selection. Both seem to involve first selecting something, and then issuing a command based on the goal:

```
Method_for_goal: Cut Selection
  Step 1. Accomplish_goal: Select Text.
  Step 2. Accomplish_goal: Issue Command using Cut.
  Step 3. Return_with_goal_accomplished.

Method_for_goal: Paste Selection
  Step 1. Accomplish_goal: Select Insertion_point.
  Step 2. Accomplish_goal: Issue Command using Paste.
  Step 3. Return_with_goal_accomplished.
```

The same similarity analysis as the previous example can be applied. However, the first step is not at all identical in the two methods - replacing the goal verb with `X` does not remove the difference between selecting text and selecting an insertion point. 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

Execution time for GOMSL statements. When the GLEAN tool executes a GOMS model, each GOMSL statement incurs a time cost that is based on the underlying production rule model. The basic rule is that each step corresponds to a single production rule; following the EPIC architecture, each rule requires 50 ms to execute. Based on the production rule model, the execution time for intrastep operators like working memory access are assumed to be bundled into this 50 ms time.

However, other operators require additional time for their activity. For example, the `Keystroke` operator involves commanding the Manual Motor Processor to generate the hand movements for the keystroke. In EPIC, the time required is calculated based on a somewhat complex model for motor movement (Kieras & Meyer, 1997). However, GLEAN uses a simple approximation based on the Keystroke-Level Model, which is that the Keystroke operator requires 280 ms. In addition, in this basic form of the GOMS model, the basic rule for executing a method is that all of the operators on each step will be executed before the next step is started (see John & Kieras, 1996b for more discussion). Thus a step involving a keystroke requires a total time of 50 ms for the step processing plus 280 ms for the keystroke, for a grand total of 330 ms.

Certain GOMSL statements correspond to procedure calls, declarations, and returns in a traditional programming language, and thus likewise require execution time for the housekeeping functions. These statements typically correspond to single production rules as well. Thus, the "overhead" time to call a method always consists of three GOMSL 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 conditionals 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 three GOMSL statement execution times, regardless of how many `If-Thens` there are in the selection rule set.

Likewise, in a steps containing a `Decide` operator, the conditionals are tested in parallel, so there is only one step execution time, no matter how many conditionals there are in the operator. If no conditions match, then a full step execution time is still required because the production rule models must still fire a rule to sequence to the next step in the method.

In summary:

- Each step requires 50 ms step execution time plus the execution time for any interstep operator.
- Method_for_accomplishing_goal and Selection_rule_set_for_goal statements require 50 ms for execution.
- The body of a selection rule set always requires 50 ms for the rule that matches plus 50 ms for the `Return_with_goal_accomplished` statement.
- A step containing a `Decide` operator requires 50 ms step execution time regardless of the number of conditionals, plus any additional execution time for interstep operators.

Time estimate calculation. The execution time for a GOMS model can only be estimated for specific task instances because only then will the number and sequence of steps and operators be determined. 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. GOMSL and GLEAN has been defined with a specified relationship to the production rule models as specified above.

GLEAN will calculate the execution time for a task instance. The calculation includes 50 ms for each production-rule cycle assumed to be involved; as listed above, a typical simple step involves one such cycle, but more can be required. In addition, GLEAN includes the Keystroke-Level Model execution times for the primitive external operators and mental operators listed above. GLEAN assumes that each mental or external operator must be completed before the next step is started, meaning that the total execution time is simply the sum of the individual operator times plus the step time. If a device simulator is included in the model, it may contribute its own waiting time. The trace and detailed output from the GLEAN tool will show the basis for the calculation. Thus, the estimated execution time is given by:

Execution Time = GOMSL Step time + Primitive External Operator Time

+ Analyst-defined Mental Operator Time + Waiting Time

GOMSL Step Time = Number of cycles × 0.05 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

For example, GLEAN produces the following output for the MacWrite example:

```
For Human: Method Execution Profile
Total Execution Time:    101.200 seconds
Freq.     Subtotal  Avg. Time  % of Total   Method for goal
   2       29.700      14.850       29.35  Copy Selection
   2       57.700      28.850       57.02  Copy Text
   1       16.200      16.200       16.01  Cut Selection
   1      101.200     101.200      100.00  Edit Document
```

```
1       6.400       6.400       6.32  Erase Text
6      54.300       9.050      53.66  Issue Command
1      30.800      30.800      30.43  Move Text
3      38.400      12.800      37.94  Paste Selection
4      95.300      23.825      94.17  Perform Unit_task
2      13.400       6.700      13.24  Select Arbitrary_text
3      10.800       3.600      10.67  Select Insertion_point
4      22.600       5.650      22.33  Select Text
2       8.800       4.400       8.70  Select Word
```

In this example, the output shows that for the set of four specified task instances, the `Issue Command` method was executed a total of 6 times, averaging about 9 s each; the total of 55.2 s was about 54% of the total time of 103 s. These times might seem to be very long; but in interpreting these times, keep in mind the basic ground rules used by GLEAN: Each external or mental operator must be completed before the next step is executed; GLEAN does not attempt to overlap operations like CPM-GOMS (Gray, John, and Atwood, 1993) or EPIC (Kieras & Meyer, 1997). Furthermore, in GLEAN all visual objects on the screen must be searched for and located - there is no credit given for familiarity in this process.

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; GLEAN reports the peak number of tags present in Working Memory, which provides a measure of how much has to be remembered at the same time. It seems reasonable to expect trouble if more than five tags have to be maintained in WM at once, at least for information that must be maintained by rehearsal in verbal working memory (see Kieras, et. al, 1999). 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.

GLEAN also includes other facilities that track the number of visual, auditory, manual, and vocal operators executed, and can produce these in the form of an execution profile for each method. Thus, if a perceptual or motor modality is especially heavily used, this aspect of workload can be identified and related to the task requirements.

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, or accesses of task information, 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 using a dummy operator like `Think_of "tab_setting_number"`. However, on a simple WYSIWYG system the user can arrive at the tab positions easily by trial and error, using operators like `Think_of "tab_setting_location"` and `Verify "setting_looks_rig` which apparently would be considerably simpler to execute than `Think_of "tab_setting_number"`. 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, and including these estimates.

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 `Recall_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.

6. NOTES ON USING THE GLEAN TOOL

Note: This section requires updating to include the specific modifications to GLEAN since it was first written. Most of it is still valid, however.

The GLEAN application is available in two forms:

- A stand-alone simulation tool that represents one simulated human interacting with a dummy device simulation. All that is required to use this tool is a GOMS model written in GOMSL in an input text file. The extended examples in this document were demonstrated with this form of the tool. The user interface for the tool is a minimal text-based interface that is compatible with all standard C++ implementations.
- A library of class definitions that support implementing your own device simulation, and constructing simulations with multiple humans and multiple devices. To use this form of GLEAN, you must be able to write Object-Oriented C++ code based on the example source files provided, using a C++ compiler and Standard Library that complies with the ANSI/ISO C++ Standard. The user interface for this form of the tool is also a minimal text-based interface, for portability.

6.1 Using the Stand-Alone Tool

A transcript of a typical MacOS session is presented below, with interspersed explanation. To get started, write your GOMSL model in a text file, execute the GLEAN application, and enter the file name at the prompt.

```
Enter GOMS model file name: :Models:MacWrite_example.gomsl
Normal output will be sent to display.
Normal, trace, detail, and debug will not be saved
Change output settings? y(es) or n(o):
```

The next prompt will ask you for modifications to the output settings. At your option, various types of output of the simulation can be saved in text files with fixed names. The normal output is the minimum to allow you to observe the model in execution and display the learning and execution results. You can save the normal output if you wish. Additional options allow you to save increasing amounts of output detail. The trace output includes in addition the sequence of steps, the events interchanged between the GLEAN architecture processors, including the simulated device, and their timings. This allows a look at the activity of the simulation, as opposed to just the results, and should be adequate for most model development and testing. The detailed output includes in addition various details such as Working Memory transactions, and the details of mouse pointing operations; it is useful primarily to help troubleshoot the model. The debug output is intended to support working at the C++ level, such as constructing a new operator or troubleshooting a custom device simulator; normally you would not save it. Quick tests of a model do not require any output to be saved, but the typical session might continue as follows:

```
Change output settings? y(es) or n(o): y
Save normal output only in file "glean.out"? y(es) or n(o): y
Save normal & trace output in file "trace.out"? y(es) or n(o): y
Save normal, trace, & detail output in file "detail.out"? y(es) or n(o): n
Save normal, trace, detail, & debug output in file "debug.out"? y(es) or n(o): n
Enter command: c, l, d, r, s, q, ?:
```

The final prompt marks the top-level command loop for the tool. Entering a '?' shows a list and brief description of the possible commands:

```
Enter command: c, l, d, r, s, q, ?: ?
Valid commands:
c(ompile) - compile method files
l(earning analysis) - produce learning time analysis
d(isplay) - show contents of GOMS model
r(un) - execute GOMS model
s(ingle step toggle) - change single step mode
q(uit) - exit
```

```
? - this list
Enter command: c, l, d, r, s, q, ?:
```

The normal next step is to compile the method file with a "c" command. The GOMSL code is processed by a parser that was constructed using the Purdue Compiler Construction Tool Set (PCCTS; Parr,1996). This made it possible to support a powerful language, but unfortunately the PCCTS-generated error messages resemble those of most compilers, and need to be interpreted using the same heuristics (e.g., the actual error may have appeared earlier than the expression the compiler is complaining about). Here is the normal output:

```
Enter command: c, l, d, r, s, q, ?: c

Compiling:Models:MacWrite_example.gomsl
Parsing done.
Model was parsed correctly.
Model is executable.
Enter command: c, l, d, r, s, q, ?:
```

Here is an example of the output produced by a syntax error, namely the tag argument for a `Look_for` operator was not a tag - it should have been `<target>` instead of `target`:

```
Enter command: c, l, d, r, s, q, ?: c

Compiling:Models:MacWrite_example.gomsl
*** look_op not found
line 154: syntax error at "target" missing TAGNAME
Found 1 syntax errors; model not built.
Parsing done.
Discovered 1 syntax errors - model is not valid!
Model is invalid due to errors.
Enter command: c, l, d, r, s, q, ?:
```

You can rapidly correct and recompile to eliminate the syntax errors. Use your favorite editor to modify the GOMSL input file, and save it to disk. The compile command will reopen the modified file and parse its contents. When an error-free compilation is obtained, you can get the learning analysis with the "l" command, see a display of the model with the "d" command, or run the model with the "r"command. The default is that the model starts in single-step mode, where GLEAN pauses before executing each step. The pause mode can be toggled off before starting execution with the "s" command. But here is how the default run starts:

```
Enter command: c, l, d, r, s, q, ?: r
*** Execution started.
Simulation starting with 10 processors
100 Human Starting: Accomplish Edit Document
150 Human Step Nil
Human->
```

The last line is a prompt for a single-step-mode command. The prior line shows the simulated real time of the pause (150 ms from the beginning of execution) and the step label (in this case `Nil`) of the step about to be executed. The item "Human" is the default name for the simulated user in this one-human simulation. The command "?" lists the possible commands:

```
Human->?
Valid commands:
n(ext) - advance to next step
g(o) - continue without pausing
d(isplay step) - show contents of this step
w(orking memory) - show contents of working memory
```

```
s(tate) - show contents of all dynamic processors
S(tatic state) - show contents of static processors
? - this list
Human->
```

Entering "n" advances GLEAN to the next step. Entering "g' will cause GLEAN to continue execution until the GOMS methods terminate (the top level method returns). It is possible to hang the execution of GLEAN, for example by waiting for the appearance of an object that never appears; you will have to force the termination of the GLEAN application to escape. The other commands support debugging the methods by showing the step about to be executed, the contents of working memory, visual working memory, the task items, and so forth. For example, after advancing several steps, we enter the text selection method required for the first task, display the first step and see that it contains a Look_for operator, execute it, and examine the results. The name of dummy visual object, `Dummy_text_word`, is now in WM under the `<target>` tag and is also in focus in the object store:

```
1750 Human Step SR Selection
Human->n
1750 Human Accomplish Select Word
1800 Human Step 1
Human->d
Step 1. Look_for_object_whose Content is Text_selection of <current_task> and_store_und
<target>.
Human->n
3100 Human Step 2
Human->w
Contents of WM tag store:
Initial contents:
Current contents:
<current_task> T1
<current_task_name> First
<target> Dummy_text_word
Current contents of WM object store:
Dummy_text_word
T1
Human->n
```

Continuing with the go command results in the output just containing the time-stamped output (in milliseconds) from the default dummy device that documents which interface actions were performed. The output concludes with some statistics on the number of Look_for operators were executed, WM usage, and the execution time profile.

```
...
98399 Device:Dummy_Device: Point_to: Paste_menu_item
99799 Device:Dummy_Device: Release: mouse_button
101350 Human:GOMS_Cognitive_Processor: No more steps to execute.
Simulation Done!
*** Execution done.

For Human: Visual Statistics
21 Visual Operations

For Human: WM Statistics
0 remaining WM_items, 15 replacements, 4 peak

For Human: Method Execution Profile
Total Execution Time:    101.200 seconds
Freq.    Subtotal  Avg. Time  % of Total   Method for goal
```

```
   2      29.700      14.850       29.35  Copy Selection
   2      57.700      28.850       57.02  Copy Text
   1      16.200      16.200       16.01  Cut Selection
   1     101.200     101.200      100.00  Edit Document
   1       6.400       6.400        6.32  Erase Text
   6      54.300       9.050       53.66  Issue Command
   1      30.800      30.800       30.43  Move Text
   3      38.400      12.800       37.94  Paste Selection
   4      95.300      23.825       94.17  Perform Unit_task
   2      13.400       6.700       13.24  Select Arbitrary_text
   3      10.800       3.600       10.67  Select Insertion_point
   4      22.600       5.650       22.33  Select Text
   2       8.800       4.400        8.70  Select Word
Enter command: c, l, d, r, s, q, ?:
```

At this point, the input file can be recompiled with the "c" command, or the model can be re-executed with the "r" command, or GLEAN can be terminated with the quit ("q") command.

6.2 Using the GLEAN Class Library for Custom Simulations

A custom simulation with multiple humans and fully interactive device simulation can be constructed using the GLEAN class library. Due to the technical detail involved, and the fact that C++ OOP expertise is required, this document does not include a detailed description of how to construct a custom simulation. Rather, a brief overview will be provided; to go further, examine the examples supplied with the library which demonstrate a variety of models.

The major effort in constructing a custom simulation is to design and implement your own device simulator to replace the default Dummy Device simulator that simply outputs a time-stamped message about whatever input it receives. The custom device will be a subclass of the `Device_Base` class, and will override the virtual functions in the `Device_Base` class that handle event messages directed to the device. For example, a device that presents a simple form to be filled in using a mouse and keyboard has the following class declaration in the file `Form_fill_in_Device.h`:

```
#ifndef FORM_FILL_IN_DEVICE_H
#define FORM_FILL_IN_DEVICE_H

#include <string>
#include "Device_Base.h"
#include "Symbol.h"
#include "Output_tee.h"

class Form_fill_in_Device : public Device_Processor {
public:
	// constructor requires processor name and output destination
	Form_fill_in_Device(const std::string& id, const Output_tee& ot) :
		Device_Base(id, ot),
		submit_button_clicked(false)
		{}

	virtual void initialize();
	virtual void display() const;
```

```
	// Overriding the following input handlers
	virtual void handle_Delay_event(const Symbol& type, const Symbol& datum,
		const Symbol& object_name, const Symbol& property_name,
		const Symbol& property_value);
	virtual void handle_Type_In_event(const Symbol& type_in_string);
	virtual void handle_Point_event(const Symbol& target_name);
	virtual void handle_Click_event(const Symbol& button_name);

private:
	Symbol keyboard_focus_field;		// target for keyboard input;
	Symbol current_pointed_to_object;	// current object being pointed to
	bool submit_button_clicked;

};

#endif
```

This example uses some classes in the GLEAN library: The `Output_tee` class is used for directing output to streams and files using the same syntax as `cout`; most of GLEAN's output is sent to one of these objects. `Symbol` implements the symbols used to represent object names, property names, and values. When the simulated user performs a mouse point, at the time the point operator completes, a Point event is generated, and this class's handle_Point_event function will be called at that time, with a parameter containing the name of the pointed-to object on the screen; the function could update the member variable `current_pointed_to_object`. Then when a `handle_Click_event` function is called, the `keyboard_focus_field` could be set to the name of the pointed-to field. One the functions provided by the Device_Base class is a `get_time()` function that can be called at any time to get the current simulated time.

Once a device simulator has been implemented, the top-level of the custom simulation is simple to construct. A `Simulation` class encapsulates all of the details of setting up a simulated human in interaction with a simulated device, and provides the command functions for running the simulation. All that is required is to create an instance of the custom device, and give it to an instance of the appropriate `Simulation` class. For example, the top-level module for the `Form_fill_in_Device` simulation is simply:

```
#include <iostream>

#include "Simulation_1H1D.h"
#include "Form_fill_in_Device.h"
#include "Output_tee.h"
#include "Glean_globals.h" // defines global output objects
using namespace std;

int main (void)
{
	// create the device
	Device_Base * device_ptr = new Form_fill_in_Device("Device", Normal_out);

	// create a simulation object, with specified model file name and device
	Simulation_1H1D simulation(":Models:form_fill_in_example.gomsl", device_ptr);

	// set up output
	simulation.setup_output();

	// start the top-level command loop
	simulation.run_top_level();
```

```
        delete device_ptr;

        return 0;

}
```

The GLEAN Library class `Simulation_1H1D` represents a simulation that has one human interacting with one device through visual/auditory and vocal/manual channels. There are several `Simulation` classes in the Library, including a two-person two-device simulation, and additional ones are easy to construct. Calling the `run_top_level` function starts the command loop for the simulation, just like the stand-alone version of GLEAN.

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; work on the GLEAN tool was originally supported by ARPA. Subsequent work on the GLEAN system was supported by occasional consulting projects sponsored by the U.S. Navy, NASA, and DARPA.

Work on GLEAN was greatly facilitated by the ease of constructing the GOMSL parser with the public-domain Purdue Compiler Construction Tool Set (PCCTS) originally developed by Terrence Parr (Parr, 1996). A good source of information about PCCTS is `http://www.mcs.net/~tmoog/pccts.html`.

REFERENCES

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

Annett, J., Duncan, K.D., Stammers, R.B., & Gray, M.J. (1971). *Task analysis*. London: Her Majesty's Stationery Office.

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.

Ericsson, K.A., & Delaney, P.F. (1999). Long-term working memory as an alternative to capacity models of working memory in everyday skilled performance. In A. Miyake & P. Shah (Eds.), *Models of working memory: Mechanisms of active maintenance and executive control*. New York: Cambridge Univerity Press. 257-297.

Ericsson, K.A., & Kintsch, W. (1995). Long-term working memory. *Psychological Review*, 102(2), 211-245.

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.

John, B. E., & Kieras, D. E. (1996a). Using GOMS for user interface design and evaluation: Which technique? *ACM Transactions on Computer-Human Interaction*, **3**, 287-319.

John, B. E., & Kieras, D. E. (1996b). The GOMS family of user interface analysis techniques: Comparison and contrast. *ACM Transactions on Computer-Human Interaction*, **3**, 320-351.

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. E. (1997a). A Guide to GOMS model usability evaluation using NGOMSL. In M. Helander, T. Landauer, and P. Prabhu (Eds.), *Handbook of human-computer interaction*. (Second Edition). Amsterdam: North-Holland. 733-766.

Kieras, D. E. (2004). Task analysis and the design of functionality. In A. Tucker (Ed.) *The Computer Science and Engineering Handbook (2nd Ed)*. Boca Raton, CRC Inc. pp. 46-1 - 46-25.

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. & Meyer, D.E. (1997). An overview of the EPIC architecture for cognition and performance with application to human-computer interaction. *Human-Computer Interaction.*, **12**, 391-438.

Kieras, D.E., Meyer, D.E., Mueller, S., & Seymour, T. (1999). Insights into working memory from the perspective of the EPIC architecture for modeling skilled perceptual-motor and cognitive human performance. To appear in A. Miyake and P. Shah (Eds.), *Models of Working Memory: Mechanisms of Active Maintenance and Executive Control*. New York: Cambridge University Press. 183-223.

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.

Kieras, D.E., Wood, S.D., & Meyer, D.E. (1997). Predictive engineering models based on the EPIC architecture for a multimodal high-performance human-computer interaction task. ACM *Transactions on Computer-Human Interaction.***4**, 230-275.

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.

Parr, T.J. (1996). *Language translation using PCCTS and C++*. San Jose, CA: Automata Publishing Co.

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.

Wood, S. (1993). Issues in the Implementation of a GOMS-model design tool. Unpublished report, University of Michigan.

Wood, S. D. (2000). Extending GOMS to Human Error and Applying it to Error-Tolerant Design. Doctoral dissertation, University of Michigan.