GOMS Models for Task Analysis

David Kieras

University of Michigan

To appear in:

D. Diaper & N. Stanton (Eds.) (in press). *Task analysis for human-computer interaction*. Lawrence Erlbaum Associates

Abstract

Analyzing a task into Goals, Operators, Methods, and Selection rules (GOMS) is an established method for characterizing a user's procedural knowledge. When combined with additional theoretical mechanisms, the resulting GOMS model provides a way to quantitatively predict human learning and performance for an interface design, in addition to serving as a useful qualitative description of how the user will use a computer system to perform a task. This chapter focusses on GOMS models as a task-analytic notation and how to construct them.

Introduction

The purpose of task analysis is to understand the user's activity in the context of the whole human-machine system, for either an existing or a future system. While understanding human activity scientifically is the goal of psychology and the social sciences, the constraints on system design activity preclude the lengthy and precise analysis and experimentation involved in scientific work. Thus a task analysis for system design must be rather more informal, and primarily heuristic in flavor compared to scientific research. The task analyst must do his or her best to understand the user's task situation well enough to influence the system design given the limited time and resources available.

Despite the fundamentally informal character of task analysis, many formal and quasi-formal systems for task analysis have been proposed. However, these systems do not in themselves analyze the task or produce an understanding of the task. Rather, they are ways to help the analyst observe and think carefully about the user's actual task activity, and provide a format for recording and communicating the results of the task analysis. Thus a task analysis methodology both specifies what kinds of task information are likely to be useful to analyze, and provides a heuristic test for whether the task has actually been understood. That is, a good test for understanding something is whether one can represent it or document it, and constructing such a representation can be a good approach to trying to understand it. A formal representation of a task helps by ensuring that the analyst's understanding is more reliably communicated. Finally, some of the more formal representations can be used as the basis for computer simulations or mathematical analyses to obtain quantitative predictions of task performance, but it must be understood that such results are no more correct than the original, and informally-obtained, understanding underlying the representation.

GOMS is such a formalized representation that can be used to predict task performance well enough that a GOMS model can be used as a substitute for much (but not all) of the empirical user testing needed to arrive at a system design that is both functional and usable. This predictive function is normally presented as the rationale for GOMS modeling (see Card, Moran, & Newell, 1983; John & Kieras, 1996a, b; Kieras, in press). However, GOMS models also qualify as a form of task-analytic representation, with properties similar to Hierarchical Task Analysis (HTA, see Annett, Duncan, Stammers, and Gray, 1971; Kirwan & Ainsworth, 1992), but with the special advantage of being able to generate useful predictions of learning and performance.

This chapter presents GOMS modeling as a task analysis method, emphasizing the process of analysis and construction of a GOMS model. Information about using the GOMS model for design evaluation and prediction of learning and performance is not covered here. GOMS methodology is quite detailed, especially when GOMS is used in a computer simulation of human performance, but due to lack of space, the presentation has been considerably simplified, and almost all specifics related to performance prediction have been eliminated. The interested reader should examine the cited sources and contact the author for treatments that are both more complete and more up to date. Also due to the lack of space, this chapter contains only one complete example of a GOMS model, a simple text editor (at the end of the chapter). The reader might find it useful to gain a preliminary understanding of what a GOMS model is like by briefly examining this example before reading further.

The GOMS Model

A GOMS model is a description of the *procedural 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.

In the Card, et al. formulation, the new user of a computer system will use various problem-solving and learning strategies to figure out how to accomplish tasks using the computer system, and then with additional practice, these results of problem-solving will become methods - procedures that the user can routinely invoke to accomplish tasks in a smooth, skilled manner. The properties of the methods will thus govern both the ease of learning and ease of use of the computer system. In the research program stemming from the original proposal, approaches to representing GOMS models based on cognitive psychology theory have been developed and validated empirically, along with the corresponding techniques and computer-based tools for representing, analyzing, and predicting human performance in human-computer interaction situations.

John & Kieras (1996a, b) 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 methods 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 (1996a,b), 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. In addition, NGOMSL models have been shown to be useful for defining the content of on-line help and documentation (Elkerton & Palmiter, 1991; Gong & Elkerton, 1990).

This chapter uses a newer computational form of NGOMSL, called **GOMSL** (**GOMS L**anguage) which is processed and executed by a GOMS model simulation tool, **GLEAN3** (**GOMS L**anguage **E**valuation and **An**alysis). GLEAN3 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), and then again as summarized in Kieras(1999). 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). GOMSL and GLEAN3 have been also been used to identifying likely sources of errors in user interfaces and model human error recovery (Wood, 1999, 2000) and also to model the performance of teams of humans who interact with speech (Santoro & Kieras, 2001).

Strengths and Limitations of GOMS Models

It is important to be clear on what GOMS models can and cannot do; see John and Kieras (1996a, b) for more discussion.

GOMS starts* after *a basic task analysis. In order to apply the GOMS technique, the task analyst must first determine what goals the user will be trying to accomplish. The analyst 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 overall goals. Approaches to this stage of interface design have been presented in sources such Gould (1988), Diaper (1989), Kirwan and Ainsworth (1992), and Beevis et al (1992). Once this basic level of task analysis has been conducted, constructing the GOMS model can then provide an elaborated account of how the user does the task.

GOMS represents only the procedural aspects of a task. GOMS models can account for the *procedural* aspects of usability; these concern the exact steps in the procedures that the user must follow, and so GOMS allows the analyst to determine 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," 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 overall 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.

General Issues in GOMS Analysis

Overview of GOMS Analysis

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 the analyst 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 top-level goals. Once a goal is defined, the corresponding method can be simple to describe 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, the details of many of these complex processes have nothing to do with the design of the interface, and so do not need to be worked out for the analysis to be useful. 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.

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, and what the natural steps are in the user's methods. These are standard problems in task analysis (see Kieras, 1997b, 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 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 the text movement task not in terms of cut and paste subgoals, but in terms of the subgoals of first selecting the text, then issuing the 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 *de facto* 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.

How do Users do the Task?

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 to get ideas about how they decompose the task in to subtasks and what methods and selection rules they use. However, 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 asking people to describe verbally what they do, a better approach is having users demonstrate on the system what they do, or better yet, observing 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 (see Bhavnani & John 1996). The analyst should try to understand why this is happening, because a good design will only be good if it is used in the intended way. But for purposes of a GOMS analysis, the analyst will have to decide whether to assume a sub-optimal use of the system, or a fully informed one.

This situation deserves further discussion. In many task-analysis or modeling situations, especially with complex systems, the human user can perform the task in a variety of different ways, following different *task strategies* - the external structure of the task does not strongly constrain what the user must do, leaving them free to devise and follow different strategies that arrive at the same end result. In such a case, it will be difficult to determine the goal structure and the methods. One approach, of course, is to rely on empirical data and observation about what users actually do in the task. This is certainly the desired approach, but it can be extremely difficult to identify the strategy people use in a task, even a very simple one (Kieras & Meyer, 2000). Furthermore, empirical results cannot be used if the system in question is under design and so has no users to observe, or if an existing system has not been, or cannot be, studied in the required detail because of the severe practical difficulties involved in collecting usage data for complex systems.

In such cases, the analyst is tempted to speculate on how users might do the task, and as noted above, such speculation by the task analyst and interface designer is likely to be better than than haphazard decisions made by whoever writes the interface code. However, if the task is indeed a complex one, trying to guess or speculate how the user does the task can result in an endless guessing-game. A better approach is to consider whether the system designers have a concept of how the system is supposed to be used, and if so, construct a GOMS model for how the user *should* do the task. This is much less speculative, and is thus relatively well-defined. It represents a sort of best-case analysis in which the system designer's intentions are assumed to be fully communicated to the user, so that the user takes full advantage of the system features. If the resulting GOMS analysis and the performance predictions for it reveal serious problems, the actual case will certainly be much worse.

An additional elaboration of this approach is to use bracketing logic (Kieras & Meyer, 2000) to gain information about the possible actual performance using the system. Construct two models of the task, one representing using the system as cleverly as possible, producing the the fastest-possible performance, and another that represents the nominal or unenterprising use of the system, resulting in a slowest-reasonable model. When performance predictions are computed, these two models will bracket what actual users can be expected to do. By comparing how the two models respond to e.g. changes in workload, and analyzing their performance bottlenecks, the analyst can derive useful conclusions about a system design, or especially about the relative merits of two designs, without having to make detailed unsupported assumptions about the user's actual task strategies (e.g. Kieras, Meyer, & Ballas, 2000).

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 (see below). 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 hardcopy. 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 below), 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.

Generative Models, Rather than Models of Specific Task 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 the analyst falls into the trap of writing methods for specific task instances, the resulting methods will probably be "flat," containing little in the way of method and submethod hierarchies, and 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, such as a different file name, means that the method will not work. 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 (for more discussion, see John & Kieras, 1996b; Kieras, in press; Kieras, Wood, & Meyer, 1997).

So, if the analyst has a collection of task scenarios or traces, he or she should study them to discover the range of things that the user has to do. They should then be set aside and a generative GOMS model written that contains a set of general methods that can correctly perform any specific task within the classes defined by the methods (e.g., delete any file whose name is specified in the task description). The methods can be checked to ensure that they will generate the correct trace for each task scenario, but they should also work for *any* scenario of the same type.

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 (1997b) 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.

GOMSL: A Notation for GOMS Models

This section presents the GOMSL (**GOMS L**anguage) notation system which is a computer-executable version of the earlier NGOMSL (Kieras, 1988, 1997a). GOMSL is an attempt to define a language that will allow GOMS models to be executed with a computer-based tool, but at the same time be easy to read. An analyst can use GOMSL in an informal fashion; if performance predictions are needed, he or she can compute the results by hand, or alternatively, tighten up the GOMSL and then use a computational tool such as GLEAN3 to run the model or generate performance predictions.

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`.

Task Data

Object-Property-Value Representation

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.

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.

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".

The object stores correspond to working memory for visual input, task information, and long-term memory retrievals. All three of these have 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. 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).

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).

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). 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.

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. Thus, these are *interstep* operators because their execution time occurs between the steps, and so governs when the next step is started.

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.

Listed below are the primitive motor and perceptual operators whose definitions and execution times are based on the physical and 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 (1990). Each operator keyword in the list below is shown 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 identifier enclosed in angle-brackets, as in *`<tag>`*, is a tag name parameter. A description of the operator and its execution time for each operator is given.

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

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

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

`Double_click` *`mouse_button`*
Two Clicks are executed. Execution time is 400 ms.

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

`Release` *`mouse_button`*
Release the mouse button. Execution time is 100 ms.

`Point_to` *`target_object`*
The mouse cursor is moved to the target object on the screen. Execution time is determined by the Welford form of Fitts' Law with a minimum of 100 ms if object location and sizes are specified; it is 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`. Execution time is 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.

`Look_for_object_whose` *`property is value, ...`* `and_store_under` *`<tag>`*
This mental operator 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.

`Get_task_item_whose` *`property`* `is` *`value`*`, ...` `and_store_under` *`<tag>`*
This mental operator is used to represent that the user has a source of information available containing the specifics of the tasks to be executed, but the analyst does not wish to specify this source, but is assuming that it requires mental activity to produce the required task information. For example, the user could be "thinking up" the tasks as he or she works, recalling it from memory, or reading the task information from a marked-up manuscript or a set of notes (see the "yellow pad" heuristic below). The task information would be specified in a set of `Task-items`, presented below, that together define a collection of objects and properties, the task description. This operator searches the task description for a task item object whose specified properties have the specified values, and stores its symbolic name in working memory under the label <tag>. It is now in focus, and additional properties are now available. Time required is 1200 ms. Putting a different task item in focus will result in the previous item's properties being no longer available. 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.

Mental operators

Mental operators are the internal actions performed by the user; they are non-observed and hypothetical, inferred by the theorist or analyst. In the notation system presented here, some mental operators are "built in;" these are primitive operators that 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`.

Below is a brief description of the GOMSL primitive mental operators.These are all intrastep operators.

Flow of control. Only one flow of control operator can appear in a step. 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 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 or complicated branching structures.

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 must 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, 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 defined:

```
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 scratchpad 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). For simplicity, 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.

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.

`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.

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 a 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.

Method Hierarchy

Methods often call sub-methods 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, a method can be called with *pseudoarguments* in the `Accomplish_goal` operator and the corresponding *pseudoparameters* can be defined for a method or selection rule set. These are automatically deleted when the method completed. 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 "pseudo" prefix makes clear that these "variables" do not follow the normal scoping rules used in actual programming languages - the human does not have a run-time function-call stack for argument passing.

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. 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.
```

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 there is 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. To control which operators in what sequence are executed within a method, use a `Decide`.

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, if the user is supposed to type in an exact string from memory, this string must be specified somehow.

The syntax for specifying auxiliary information is based on describing object-like entities with properties and values; these descriptions can appear 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. GLEAN3 reserves two property names, `Location` and `Size`, for use by the Visual and Manual processors. All other property names and values can be chosen by the analyst.

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.
```

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.
   ...
...
```

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, start by describing a method for accomplishing a top-level goal in terms of high-level operators. Then choose one of the high-level operators, replace it with an `Accomplish_goal` operator for the corresponding goal, and then write a method for accomplishing that goal in terms of lower-level operators. Repeat with the other level operators. Then descend a level of analysis, and repeat the process for the lower-level operators. Continue until the methods have arrived at enough detail to suit the design 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.

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, similar methods are more likely to be noticed, which is critical to capturing the procedural consistency of the user interface.

Step 1: Choose the top-level user's goals

The top-level user's goals are the first goals that will be expanded in the top-down decomposition. It is worthwhile to make the topmost goal, and the first level of subgoals, very high-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 `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.

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 only one of these applications is under design, say the page-layout editor, and the analysis start only with goals that correspond to page-layout functions, the analysis may miss what the user has to do to integrate the use of the page-layout editor in the rest of the environment.

As a lower-level 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 the analysis starts with `revise document` it is possible to see that one kind of revision is changing a piece of text to another, and so this especially handy method might well be noticed in the analysis. But if the analysis starts with goals like `insert text` and `delete text` the decision has already been made about how revisions will be done, and so it is more likely to miss a case where a natural goal for the user has been well-mapped onto the software directly, instead of going through the usual functions.

Step 2. Write the Top-Level Method Assuming a Unit-Task Control Structure

Unless there is reason to believe otherwise, assume that the overall task has a unit-task type of control structure. This means that the user will accomplish the topmost goal (the overall task) by doing a series of smaller tasks one after the other. The smaller tasks correspond to the set of top-level goals chosen in Step 1. For a system such as a text editor, this means that the topmost 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 GOMSL 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.

In this example the task type maps directly to a goal whose name is a near-synonym of the type, but this is not always the case. A good exercise is to consider the typical VCR, which has at least three modes for recording a broadcast program; the selection rule for choosing the recording method consists not of tests for a simple task types like "one button recording", but rather the conditions under which each mode can or should be applied. For example, if the user is present at the beginning of the program, but will not be present at the end, and the length of the program is known, then the one-button recording method should be used.

Step 3. Recursively Expand the Method Hierarchy

This step consists of writing a method for each goal in terms of high-level operators, and then replacing the high-level operators with another goal/method set, until the analysis has worked down to the finest grain size desired. First, draft a method to accomplish each of the current goals. 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 as it is developed. 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. A recommendation: defer consideration of minor alternative methods until later; especially for alternative "shortcut" methods.

After drafting all of the methods at the current level, examine them one at time. 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 the decision is whether additional detail is needed for design purposes. For example, early in the design of a specialized text-entry device, 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, it is a good idea 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. For each operator to be expanded, rewrite that step in the method (and in all other methods using the operator) to replace the operator with an `Accomplish_goal` operator for the corresponding 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.
```

To descend a level of analysis for the Step 1 operator `Select Text`, 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 provide a method for the goal of selecting the text. This process should be repeated until all of the methods consist only of operators that are either primitive operators, or higher-level operators that will not be expanded.

Step 4. Document and Check the Analysis

After the methods and auxiliary information has been written out to produce the complete GOMSL model, list the any analyst-defined operators used, along with a brief description of each one, and the assumptions and judgment calls 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, and correct and recheck if necessary.

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. It may be desirable to develop alternate GOMS models to capture the effects of different judgment calls to systematically evaluate whether they have important impacts on the design.

Example in the Appendix

The Appendix contains a complete example GOMS model and a summary of how it was constructed with the above procedure. A more complete description of the construction procedure can be found in Kieras (1988, 1997a, 1999).

Conclusion and New Directions

GOMS was originally intended as an analytic approach to evaluating a user interface: a way to obtain some usability information early enough in the design process to avoid the expense of prototype development and user testing (see John & Kieras, 1996a,b; Kieras, in press). However, as GOMS was developed, it incorporated some of the developing ideas about computational modeling of human cognition and performance, and has thus become a framework for constructing computer simulations of the subset of human activity that is especially relevant to much of user interface design. Thus, as presented here, GOMS can be defined as a task-analytic notation for procedural knowledge that (1) has a syntax and semantics similar to a traditional programming language; (2) assumes a simplified cognitive architecture based on the scientific literature that can be implemented as a computer simulation; (3) can be executed in a simulation to yield a simulated human that can interact with an actual or simulated device; (4) the static and run-time properties of the resulting simulation model predict aspects of usability

such as time to learn or task execution time.

The advantage of GOMS modeling of human activity over the several alternatives (see Kieras, in press) is its relative simplicity and conceptual familiarity to software designers and developers. The key part of such simulations is, of course, the representation of how the simulated human understands the task structure and the requirements; GOMS models provide a convenient and relatively intelligible way to describe the procedural aspects of tasks. The fact that these task-analytic models have both a direct scientific tie to human psychology and also can be used in running computer simulations means that the GOMS task-analytic notation has well-grounded claims to rigor and utility beyond more informal approaches. Of course, formality has a price: constructing a formal representation is always more work than an informal one. However, GOMS models can be "sketched" in an informal manner that preserves the intuitive concepts of how the task is done, and these informal models can then be tightened up if necessary. Thus there would appear to be no disadvantages to using GOMS for representing procedural tasks: the level of formality of the notation can be adjusted to the requirements of the analysis situation.

Although GOMS as it currently stands is a useful and practical approach, it is by no means "finished." There are many unresolved issues and new directions to explore within the general approach that GOMS represents, and the specifics of GOMS as presented in this chapter. The remainder of this concluding discussion will deal with three topics undergoing development in GOMS; these are human error; interruptions, and modeling of teams.

Modeling Error Behavior

The GOMS models originally presented in Card, Moran, and Newell (1983) and subsequently dealt only with *error-free* behavior, although Card et al (1983, p. 184ff) did sketch out what would be required to apply GOMS to errors. Historically, it has been a daunting problem to model human error and how to deal with it in design more precisely and specifically than the usual high-level general advice. However, as presented at length in Wood (2000), if attention is restricted to errors in procedural tasks, and GOMS is used to represent procedural knowledge, substantial progress can be made. That is, a remarkable thing about the extant theoretical work on human error is that it does not have at its core a well-worked out and useful theory of normal, or error-free, behavior in procedural tasks; it is hard to see how one could account for errors unless one can also account for correct performance! GOMS, in the architectural sense presented here, is such a theory of correct performance, and thus provides good starting point for usefully representing human error behavior.

Wood (2000) follows up on this insight and the original Card et al. proposal in three general ways: First, once the human detects an error, recovering from it becomes simply another goal; a well-designed system will have simple, efficient, and consistent methods for recovering from errors. Thus the error-recovery support provided by a user interface can be designed and evaluated with GOMS just like the "normal" methods supported by the interface. Second, the way in which an error recovery method should be invoked turns out to be a difficult notational problem, and has a direct analogy to how error handling is done in computer programming. The modern solution for computer programming is *exceptions*, which provide an alternate flow of control to be used just for error handling, leaving the main body of the code uncluttered to represent only the normal activity of the program. Wood suggests that a similar approach would be the desirable extension to GOMS models: when an error is detected, an exception-like mechanism invokes the appropriate error-recovery method and then allows the original method to resume. However, exception handling is subtle even in computer programming, and humans may or may not work in the same way; more theoretical and empirical work is needed. Third, the ways in which humans detect that they have committed an error is currently rather mysterious, and we lack a good theoretical proposal that could be used easily in design situations. To finesse this problem, Wood (1999, 2000) proposed using a set of heuristics for examining a GOMS model and identifying what type of error was likely to occur at each method step, when the error would become visible, and what method the user would have to use to recover from it. This information in turn suggests how one might modify the interface to reduce the

likelihood of the error or make it easier to detect and recover from it. For example, suppose the GOMS model includes a mental operation to compute a value from two numbers on the screen that is subsequently used in a decide operator that invokes one of two submethods to complete a task. A possible error is to miscompute the value, but the error might not be manifested until the decide operator had taken the user through a series of other steps to the wrong display, for which no further steps could be executed. Such an interface, with its error-prone requirements and delayed-detection properties, might compound its poor design by forcing the user to start from the beginning to correct the error. The interface could be redesigned to make the computation unnecessary if possible, make an error obvious sooner, or provide an efficient recovery procedure. Wood (2000) demonstrated the value of this heuristic analysis for error-tolerant design using a realistic e-commerce application.

Interruptions

The normal flow of control in a GOMS model as presented here is the hierarchical-sequential flow of control used in traditional programming languages: a method executes steps in sequence; if a step invokes a submethod, the steps in that submethod are executed in sequence until the submethod is complete, whereupon the next step in the calling method is executed in sequence. This simple control regime is why GOMS models are relatively easy to construct compared to other cognitive modeling approaches (see Kieras, in press). However, in realistic situations, humans often have to respond to interrupts of various kinds; an everyday example is responding to a telephone call, then resuming work after handling the call. Trying to account for interruptability within the confines of hierarchical-sequential flow of control is technically possible, but it is also clumsy and counter-intuitive: statements that check for interrupting events must be distributed liberally into the normal flow of processing. Computer technology rejected such an approach many decades ago with the introduction of specialized hardware in which an interrupting signal automatically forces the computer to suspend execution of whatever it is doing and start executing interrupt-handling code instead; once the interruption is dealt this, the interrupted process can be resumed.

GOMS models for many computer applications do not seem to require such interrupt processing because (1) the analyzed task is limited to the human interacting with the computer; other devices, such as the telephone, are not included; and (2) the activity with the computer is all user-initiated; the computer responds only to the user's activity, and in such a way that the user always waits for the computer to finish its response before continuing. The typical text-editor task fits this description, along with many ordinary computer-usage situations. However, in other tasks, the machine can present events that are *asynchronous* with the user's activity, and the task requirements can be such that the user must respond to these asynchronous events promptly, or at least not ignore them. An example of such a task situation appears in the military task modeled by Santoro, Kieras, and Campbell (2000). Here the user is supposed to monitor a radar display showing the movements of aircraft in the vicinity of a warship and perform various tasks in response to what the aircraft do. For example, if an aircraft exhibits suspicious behavior, the user is supposed to establish radio contact with the aircraft and ask for identification and clarification. Such activity can take several minutes to complete; but in the meantime, other events must be noted, even if no overt activity is performed in response. For example, another aircraft could suddenly appear on the display, and the user must note that it should be given priority for inspection and decision-making once the current activity is done. In analogy with computer programming, such checking could be done with many statements throughout the GOMS methods, but both practicality and intuition requires some kind of interrupt mechanism analogous to those used in computers. Some of the production-rule cognitive architectures (see Byrne, in press) provide a natural approach: GOMS can be extended to include a set of If-Then statements whose conditions are evaluated whenever the relevant psychological state of the user changes; these rules specify what goal to accomplish if a specific interrupting condition is present. Thus for the radar operator's task, one interrupt rule was that if a new "blip" with a red color-code appears on the display, add it to a list of high-priority blips. As part of its process for choosing the next task, the top-level method in the model checks this list and activates a goal based on what it finds there.

The interrupt-rule concept provides a natural mechanism for giving a GOMS model the ability to respond to asynchronous events while preserving the simplicity of the hierachical-sequential control structure for the bulk of the task. It also provides a potential way to represent error detection; an interrupt rule could be checking for evidence of an error, and then invoke the appropriate error-handling method. Working out the details of such an approach is a matter for future research. But in the meantime, it appears that GOMS models can successfully combine a simple program-like representation with an intuitive form of interruptability.

Modeling teams of users

Many design situations involve teams of humans that cooperate to perform a complex task. Doing more than simply acknowledging the possible incredible complexity of human interactions involved in a team is well beyond the scope of this chapter. However, there is a subset of team activity that can be encompassed with GOMS modeling: the case where the team is following a procedure consisting of specified interactions between the team members, each of whom is likewise following a set of specified procedures. Such situations are common in many military team situations, such as the *combat information center* teams analyzed by Santoro and Kieras (2001). For example, each human in the team sits at a workstation that incorporates a radar display, and has certain assigned tasks, such as making the radio contacts described above. The team members are supposed to communicate with each other, using speech over an intercom, to coordinate their activity, such as ensuring that high-priority blips get examined. In this case, a model for the team can be constructed simply as a team of models: Each team member's taks is represented by a GOMS model; part of the member's task is to speak certain messages to other team members, and respond to certain messages from other team members. The structure of the team procedures determines which messages are produced by what member, and how another member is supposed to respond to them. The interrupt capability described above is especially useful because it simplifies handling of asynchronous speech input. Once the individual GOMS models have been developed, the activity of a team can be simulated simply by running the whole set of interacting individual models simultaneously. The simulation can then show whether the team procedures result in good performance by the team as a whole. Thus the rigor and strengths of GOMS modeling and task analysis can be extended from the domain of individual user interfaces to the domain of team structure and team procedures.

Appendix: An Example

Below is shown a complete example GOMS model for a simplified subset of the MacWrite text editor that describes how to move, delete, or copy text selections. The model contains methods that start at the topmost level and finish at the keystroke level, along with the necessary auxiliary information for the methods to be executed (by hand or by GLEAN), and a set of four benchmark tasks to be performed. The GOMSL starts with the auxiliary information for a set of tasks, visual items, and items assumed to be in LTM. The methods themselves start with the Top-Level Unit Task method. One should be able to get at least of rough idea of the methods simply by reading them, even without detailed knowledge of the syntax of GOMSL; this is one of the goals of the NGOMSL and GOMSL notations.

The process of constructing the model will be summarized. A more complete step-by-step construction of a similar example can be found in Kieras (1988, 1997a, 1999). Due to the top-down expansion of methods, the methods were constructed in roughly the same order as they appear in the example. The construction started with the topmost user's goal of editing the document. Taking the above recommendation, the first piece of the model is simply a version of the unit-task method and the selection rule set that dispatches control to the appropriate method. This assumes an ad-hoc task representation that was refined as the construction continued.

The unit task method and its selection rule specify a set of second-level user goals. This example focusses on the methods for the goal of moving text. A first judgment call was that users view moving text as first cutting, then pasting. The method for this goal was written accordingly, initially with two high level operators: `Cut Selection`, then `Paste Selection`, followed by a `Verify` that the cut and paste has been done correctly.

Descending a level, the high-level operators were then rewritten into `Accomplish_goal` operators and methods for cutting and pasting selected text were provided. Another judgment call is that users are aware of the general-purpose idea of text selection, so the method for cutting a selection starts not with keystroke-level actions for selecting text that would be specific to the cutting goal, but rather with another high-level operator for selecting text which then was rewritten into an `Accomplish_goal` operator. Since there are a variety of ways to select text, a selection rule specifies three different specific contexts in which text selection is needed. It thus maps the general goal of selecting text to three different specific goals, each of which has its own method. The paste method similarly has a subgoal of selecting the insertion point, but there is only one way to do it, and so only a single method was provided. Notice how this set of judgements effectively state that the user has some general-purpose "subroutines" for cutting, pasting, selecting text, and selecting an insertion point. Expressing this conclusion as goals and methods asserts some key properties of the interface (e.g. selection can be done in the same way for all relevant tasks in the text-editing application) and that the user makes use of them, or *should* make use of them.

Descending another level, note that the cutting and pasting methods involve picking a command from a menu (for simplicity, assume that users do not make use of the command-key shortcuts). An important property of well-designed menu systems is that the procedure for picking a command is uniform across the menu system. Thus, the operators of `Invoke_cut_command` and `Invoke_paste_command` were replaced with a single `Issue Command` method that is given the "name" or "concept" of the desired command as a pseudoargument and makes the proper menu accesses. Thus the `Issue Command` method first retrieves from LTM which menu to open, finds it on the screen, opens it, and then finds and selects the actual menu item.

To make this example more complete, the methods for deleting and duplicating text were 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; this is one symptom of a good design. After drafting the methods the analyst collected the task information that the methods require, and reconciled and revised the task representation as necessary. In addition, the auxiliary information was collected and specified, such as the LTM items required by the `Issue Command`

method. For the GLEAN tool to execute these methods, there needs to be some visual objects for the methods to look for and point at. In this case, these objects need only be minimal or "dummy" objects. The example also includes as auxiliary information a set of four editing tasks specified in a "linked list" form that is accessed by the top-level unit task method.

```
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.
```

References

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.

Beevis, D., Bost, R., Doering, B., Nordo, E., Oberman, F., Papin, J-P., I., H. Schuffel, & Streets, D. 1992. Analysis techniques for man-machine system design. (Report AC/243(P8)TR/7). Brussels, Belgium: Defense Research Group, NATO HQ.

Bhavnani, S. K., & John, B. E. (1996). Exploring the unrealized potential of computer-aided drafting. In *Proceedings of the CHI'96 Conference on Human Factors in Computing Systems,* ACM, New York, 1996.

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.

Byrne, M. D. (in press). Cognitive architecture. In J. Jacko & A. Sears (Eds), *Human-Computer Interaction Handbook*. Mahwah, N.J.: Lawrence Erlbaum Associates

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

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.

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. (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. (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. (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. (1997b). Task analysis and the design of functionality. In A. Tucker (Ed.) *The Computer Science and Engineering Handbook*. Boca Raton, CRC Inc. 1401-1423.

Kieras, D.E. (1999). *A Guide to GOMS Model Usability Evaluation using GOMSL and GLEAN3*. Document available via anonymous ftp at ftp://www.eecs.umich.edu/people/kieras

Kieras, D. E. (in press). Model-based evaluation. In J. Jacko & A. Sears (Eds), *Human-Computer Interaction Handbook*. Mahwah, N.J.: Lawrence Erlbaum Associates

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. (2000). The role of cognitive task analysis in the application of predictive models of human performance. In J. M. C. Schraagen, S. E. Chipman, & V. L. Shalin (Eds.), *Cognitive task analysis*. Mahwah, NJ: Lawrence Erlbaum, 2000.

Kieras, D., Meyer, D., & Ballas, J. (2001). Towards demystification of direct manipulation: Cognitive modeling charts the gulf of execution. *Proceedings of the CHI 2001 Conference on Human Factors in Computing Systems*. New York, ACM. Pp. 128 – 135.

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. 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.

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

Olson, J. R., & Olson, G. M. (1990). The growth of cognitive modeling in human-computer interaction since GOMS. *Human-Computer Interaction*, 5, 221-265.

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.

Santoro, T.P., Kieras, D.E., and Campbell, G.E. (2000). GOMS modeling application to watchstation design using the GLEAN tool. *Proceedings of the Interservice/Industry Training, Simulation, and Education Conference*, pp. 964-973, Orlando, FL. November, 2000.

Santoro, T., & Kieras, D. (2001). GOMS models for team performance. In J.Pharmer and J. Freeman

(Organizers), Complementary methods of modeling team performance. Panel presented at The 45th Annual Meeting of the Human Factors and Ergonomics Society, Minneapolis/St. Paul.

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

Wood, S. D. (1999). The Application of GOMS to Error-Tolerant Design. Paper presented at the 17th International System Safety Conference, Orlando, FL.

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